by robin wieruch

the Road to learn React

The Road to learn React (Spanish)

Tu camino para aprender React plano y pragmático

Robin Wieruch, Juan Luis Chaurant, Emanuel Canova y Guillermo Salazar

Este libro está a la venta en http://leanpub.com/the-road-to-learn-react-spanish

Esta versión se publicó en 2018-12-07

Este es un libro de Leanpub. Leanpub anima a los autores y publicadoras con el proceso de publicación. Lean Publishing es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener feedback del lector hasta conseguir tener el libro adecuado.

© 2017 - 2018 Robin Wieruch, Juan Luis Chaurant, Emanuel Canova y Guillermo Salazar

¡Twitea sobre el libro!

Por favor ayuda a Robin Wieruch, Juan Luis Chaurant, Emanuel Canova y Guillermo Salazar hablando sobre el libro en Twitter!

El tweet sugerido para este libro es:

I am going to learn #ReactJs with The Road to learn React by @rwieruch Join me on my journey https://roadtoreact.com

El hashtag sugerido para este libro es #ReactJs.

Descubre lo que otra gente está diciendo sobre el libro haciendo click en este enlace para buscar el hashtag en Twitter:

#ReactJs

Índice general

Prefacio

The Road to learn React te enseñará los fundamentos de React. Construirás una aplicación con funcionalidades reales utilizando solo React, sin ninguna otra complicada herramienta. Todo, desde la configuración de tu proyecto hasta como subirlo a un servidor está explicado. El libro incluye material adicional y ejercicios en cada capítulo. Después de leer el libro, estarás en capacidad de construir tus propias aplicaciones en React. El material es constantemente actualizado por mi y la comunidad.

En the Road to learn React, primero obtienes los fundamentos básicos antes de sumergirte dentro del extenso ecosistema React. Los conceptos serán explicando pocas herramientas, poca gestión de estados y mucha información sobre React. Se explican conceptos generales, patrones y ejercicios prácticos dentro del contexto de una aplicación real.

Esencialmente, aprenderás a construir tu aplicación React desde los cimientos, con características como paginación, almacenamiento en el lado del cliente (client-side caching), e interacciones de búsqueda y clasificación. Harás la transición de JavaScript ES5 a JavaScript ES6. Espero que este libro capture mi entusiasmo por React y JavaScript y te ayude a iniciarte en estas tecnologías.

Acerca del Autor

Soy un ingeniero de software Alemán dedicado al aprendizaje y la enseñanza de programación en JavaScript. Después de obtener mi Master en ciencias de la computación, seguí mi aprendizaje por cuenta propia. Obtuve experiencia del mundo de las startups, donde utilize JavaScript intensivamente durante mi vida profesional y mi tiempo recreacional, eventualmente esto se transformo en pasión por transmitir a otros mis conocimientos sobre estos temas.

Por algunos años, trabajé de cerca con un excepcional equipo de ingenieros en una compañía llamada Small Improvements, desarrollando aplicaciones a larga escala. La compañía ofrecía un producto SaaS que permite a los usuarios proveer opiniones directamente a los negocios. Esta aplicación se desarrolló utilizando JavaScript en el frontend y Java como backend. La primera iteración del frontend de Small Improments se escribió en Java con el framework Wicket y jQuery. Cuándo la primera generación de SPAs se hizo popular, la compañía migró el frontend de la aplicación a Angular 1.x. Luego de utilizar Angular por aproximadamente dos años, se hizo claro que este framework no era la mejor solución para trabajara con aplicaciones que hacen uso intenso de los estados, así que fue necesario migrar de Angular a React y Redux. Esto permitió que la aplicación operara de manera exitosa a larga escala.

Durante mi tiempo en la compañía, escribí artículos de manera regular sobre desarrollo web para mi sitio. Recibí gran feedback de personas que estaban aprendiendo directamente de mis artículos, lo que me permitió mejorar mi escritura y mi estilo de enseñanza. Con cada artículo nuevo, mi confianza para enseñar a otros fue creciendo. Sentía que mis primeros artículos contenían demasiada información, abrumador para muchos estudiantes, pero lo mejoré al enfocarme en solo un tema a la vez.

Actualmente, soy un ingeniero de software independiente y educador. Me parece gratificante ver como los estudiantes crecen cuando se les da objetivos claros y retroalimentación concisa. Puedes leer más acerca de mi y apoyar mi trabajo en mi sitio web[1].

[1] https://www.robinwieruch.de/about

Testimonios

Hay muchos testimonios[2], calificaciones[3] y opiniones[4] sobre el libro que puedes leer en línea. Estoy orgulloso de el y nunca esperé tan arrollador feedback. Me encantaría leer tu clasificación/opinión. Me ayuda a decirle al mundo acerca del libro y estar listo para futuros proyectos. A continuación algunas opiniones:

Muhammad Kashif[5]: "The Road to Learn React es un libro único que recomiendo a cualquier estudiante o profesional interesado en aprender desde conceptos básicos de React hasta un nivel avanzado. Está lleno de consejos y técnicas interesantes que son difíciles de encontrar en otros lugares, y notablemente completo en su uso de ejemplos y referencias a problemas de muestra, tengo 17 años de experiencia en el desarrollo de aplicaciones web y de escritorio, y antes de leer este libro tuve problemas para aprender React, pero este libro funciona como magia."

Andre Vargas[6]: "The Road to Learn React por Robin Wieruch es un gran libro. Todo lo que se acerca de React lo aprendí de el."

Nicholas Hunt-Walker, Instructor de Python en una Escuela de Programación de Seattle:[7]: "Este es uno de los libros de programación mejor informados y bien escritos que utilizado. Una introducción sólida a React y ES6."

Austin Green[8]: "Gracias, realmente disfruté el libro. Perfecta mezcla para aprender React, ES6 y conceptos de programación de alto nivel."

Nicole Ferguson[9]: "I'm doing Robin's Road to Learn React course this weekend & I almost feel guilty for having so much fun."

Karan[10]: "Just finished your Road to React. Best book for a beginner in the world of React and JS. Elegant exposure to ES. Kudos! :)"

Eric Priou[11]: "The Road to learn React by Robin Wieruch is a must read. Clean and concise for React and JavaScript."

[2] https://roadtoreact.com/
[3] https://www.goodreads.com/book/show/37503118-the-road-to-learn-react
[4] https://www.amazon.com/dp/B077HJFCQX
[5] https://twitter.com/appsdevpk
[6] https://twitter.com/andrevar66/status/853789166987038720
[7] https://github.com/nhuntwalker
[8] https://twitter.com/AustinGreen/status/845321540627521536
[9] https://twitter.com/nicoleffe/status/833488391148822528
[10] https://twitter.com/kvss1992/status/889197346344493056
[11] https://twitter.com/erixtekila/status/840875459730657283

Educación para niños

Este libro es de código abierto y debería permitir a todos aprender React. Sin embargo, no todo el mundo tiene el privilegio de acceder a los recursos requeridos, principalmente porque no todos saben Inglés. Quiero utilizar este proyecto como soporte para otros proyectos que ayuden a enseñar Inglés a niños que viven en Países en desarrollo.

Del 11 de Abril al 18 de Abril, 2017, Devolviendo, Aprendiendo React[12]

[12] https://www.robinwieruch.de/giving-back-by-learning-react/

Preguntas frecuentes

¿Cómo consigo actualizaciones? Puedes suscribirte[13] al boletín o seguirme en Twitter[14] para recibir actualizaciones. Una vez que tenga una copia del libro, se mantendrá actualizado cuando se lance una nueva edición. Pero tienes que descargar una copia nuevamente cuando se anuncie una actualización.

¿Utiliza la versión reciente de React? El libro siempre recibe una actualización cuando se actualiza la versión de React. Por lo general, los libros están desactualizados poco después de su lanzamiento. Dado que este libro es auto publicado, puedo actualizarlo cuando lo desee.

¿Cubre Redux? No. El libro debe proporcionarle una base sólida antes de sumergirse en temas avanzados. La implementación de la aplicación de ejemplo en el libro mostrará que no necesita Redux para construir una aplicación en React. Después de haber leído el libro, debería poder implementar una aplicación sólida sin Redux.

¿Utiliza JavaScript ES6? Sí. Pero no te preocupes. Estarás bien si estás familiarizado con JavaScript ES5. Todas las características de JavaScript ES6 que describo en el viaje para aprender React cambiarán de ES5 a ES6 en el libro. Se explicarán todas las características en el camino. El libro no solo enseña React, sino también todas las funciones útiles de JavaScript ES6 para React.

¿Agregará más capítulos en el futuro? Puedes consultar el registro de cambios para ver las actualizaciones más importantes que ya sucedieron. Habrá mejoras no anunciadas en el inter también. En general, depende de la comunidad si continúo trabajando en el libro. Si hay una aceptación para el libro, entregaré más capítulos y mejoraré el material antiguo. Mantendré el contenido actualizado con las mejores prácticas, conceptos y patrones recientes.

¿Cuáles son los formatos de lectura? Además de los formatos .pdf, .epub y .mobi, puedes leerlo en markdown desde GitHub[15]. En general, recomiendo leerlo en un formato adecuado, de lo contrario, los fragmentos de código tendrán feos saltos de línea.

¿Cómo puedo obtener ayuda mientras leo el libro? El libro tiene un Canal de Slack[16] para las personas que están leyendo el libro. Puedes unirte al canal para obtener ayuda o para ayudar a otros. Después de todo, ayudar a otros también puede mejorar sus aprendizajes.

¿Hay algún área de solución de problemas? Si tienes problemas, únete al canal de Slack. Además, puede echar un vistazo a los issues en GitHub[17] sobre el libro. Quizás el problema ya fue mencionado y puede encontrar la solución para ello. Si el problema no fue mencionado, no dude en abrir un nuevo issue donde pueda explicar su problema, tal vez proporcionar una captura de pantalla y algunos

[13] https://www.getrevue.co/profile/rwieruch
[14] https://twitter.com/rwieruch
[15] https://github.com/rwieruch/the-road-to-learn-react
[16] https://slack-the-road-to-learn-react.wieruch.com/
[17] https://github.com/rwieruch/the-road-to-learn-react/issues

detalles más (página del libro, versión de node). Después de todo, trato de enviar todos los arreglos en la siguiente edición del libro.

¿Por qué el libro es paga lo que quieres? He puesto mucho esfuerzo en esto y lo haré en el futuro. Mi deseo es llegar a tantas personas como sea posible. Todos deben estar capacitados para aprender React. Aún así podrías pagar, si puedes pagarlo. Además, el libro intenta apoyar proyectos que educan a niños en paises en desarrollo[18]. Tu también puedes tener un impacto.

¿Puedo ayudar a mejorarlo? Sí. Puedes tener un impacto directo con tus pensamientos y contribuciones en GitHub[19]. No pretendo ser un experto ni escribir en inglés nativo. Apreciaría mucho tu ayuda.

¿Puedo apoyar el proyecto? Sí. Siéntete libre de hacerlo. Invierto mucho de mi tiempo en tutoriales de código abierto y recursos de aprendizaje. Puedes echar un vistazo a la sección acerca de mi[20] de mi página.

¿Hay un llamado a la acción? Sí. Quiero que te tomes un momento para pensar en una persona que le gustaría aprender React. La persona podría haber mostrado interés ya, podría estar en medio de aprender React o puede que todavía no esté consciente de querer aprender a React. Ponte en contacto con esa persona y compartele el libro. Significaría mucho para mí. El libro está destinado a ser compartido con otros.

[18] https://www.robinwieruch.de/giving-back-by-learning-react/
[19] https://github.com/rwieruch/the-road-to-learn-react
[20] https://www.robinwieruch.de/about/

Registro de Cambios

10. Enero 2017:

- v2 Pull Request[21]
- aún más amigable para principiantes
- 37% más de contenido
- 30% de contenido mejorado
- 13 capítulos mejorados y nuevos.
- 140 páginas de material de aprendizaje.
- + curso interactivo de educative.io[22]

08. Marzo 2017:

- v3 Pull Request[23]
- 20% más de contenido
- 25% de contenido mejorado
- 9 nuevos capítulos.
- 170 páginas de material de aprendizaje.

15. Abril 2017:

- Actualización a React 15.5

5. Julio 2017:

- Actualización a node 8.1.3
- Actualización a npm 5.0.4
- Actualización a create-react-app 1.3.3

17. October 2017:

- Actualización a node 8.3.0
- Actualización a npm 5.5.1
- Actualización a create-react-app 1.4.1

[21] https://github.com/rwieruch/the-road-to-learn-react/pull/18
[22] https://www.educative.io/collection/5740745361195008/5676830073815040
[23] https://github.com/rwieruch/the-road-to-learn-react/pull/34

- Actualización a React 16
- v4 Pull Request[24]
- 15% más de contenido
- 15% de contenido mejorado
- 3 capítulos nuevos (Bindings, Event Handlers, Error Handling)
- 190+ páginas de material de aprendizaje
- +9 códigos fuente de proyectos[25]

17. February 2018:

- Actualización a node 8.9.4
- Actualización a npm 5.6.0
- Actualización a create-react-app 1.5.1
- v5 Pull Request[26]
- Más caminos de aprendizaje
- Material de aprendizaje extra
- 1 capítulo nuevo (Axios instead of Fetch)

31. August 2018:

- Revisión y edición profesional a la versión en Inglés, por Emmanuel Stalling
- 16 códigos fuente de proyectos[27]
- v6 Pull Request[28]

3. October 2018:

- Actualización a node 10.11.0
- Actualización a npm 6.4.1
- Actualización a create-react-app 2.0.2

[24] https://github.com/rwieruch/the-road-to-learn-react/pull/72
[25] https://roadtoreact.com
[26] https://github.com/the-road-to-learn-react/the-road-to-learn-react/pull/105
[27] https://roadtoreact.com
[28] https://github.com/the-road-to-learn-react/the-road-to-learn-react/pull/172

Reto

Escribo bastante sobre las cosas que aprendo. Así es como llegué a donde estoy ahora. La mejor manera de enseñar un tema es justo después de que lo has aprendido. Enseñar me ha ayudado mucho en mi carrera y quiero que tu experimentes los mismos efectos. Pero primero tienes que enseñarte a ti mismo. Mi reto para ti es el siguiente: Enséñale a otras personas sobre lo que aprendas mientras lees el libro. Algunas ideas acerca de cómo puedes lograr esto:

- Escribe un artículo acerca de un tema específico encontrado en el libro. No se trata de copiar y pegar el material, en vez, intenta enseñar el tema con tus propias palabras. Toma un problema específico y resuélvelo, profundiza más en el tema tratando de entender cada detalle. Entonces, enséñalo a otros por medio de tu artículo. Verás como llenas tus brechas de conocimiento, porque al crear tu propia opinión te esfuerzas en profundizar en el tema y así, se abren las puertas de tu carrera a largo plazo.
- Si eres activo en redes sociales, toma un par de puntos que hayas aprendido mientras leías el libro y compártela con tus amigos. Por ejemplo, puedes tomar una captura de pantalla del libro y twitearlo, o mejor aún: Escribe acerca del tema con tus propias palabras, de esta manera puedes enseñar sin necesidad de invertir demasiado tiempo en ello.
- Si te sientes confiado grabando tu aventura de aprendizaje, comparte tus aprendizajes a través de Facebook Live, Youtube Live o Twitch. Te ayuda a mantenerte concentrado y completar los objetivos a lo largo del libro. Aunque no hayan muchas personas siguiendo tu transmisión, siempre puedes usar el video para cargarlo a YouTube. Además, es una gran forma de verbalizar tus problemas y la manera en que los solventas.

Me encantaría ver a las personas realizando especialmente el último punto: Grabarse a si mismos al momento de leer el libro, implementando las aplicaciones y resolviendo los ejercicios para luego poner la versión final en YouTube. Si algo está tomando más de la cuenta, puedes editar el video o utilizar un efecto para avanzar en cámara rápida. Si encuentras una dificultad que no puedas superar, no lo dejes fuera del video, esto es muy valioso para tu audiencia, que probablemente encontrará dificultades similares a las tuyas. Un par de tips para el video:

- Hazlo en tu idioma nativo o en Inglés si te sientes cómodo con ello.
- Verbaliza tus ideas, lo que estas haciendo o los problemas que intentas resolver. Tener un video visual es solo una parte del reto, la otra parte es narrar lo que haces mientras implementas el código. No tiene que ser perfecto. Por el contrario, debe sentirse natural, no como en otros videocursos donde nadie puede ver los problemas que se presentan de improvisto.

Si consigues algunos bugs o problemas, no te sientas mal por ello. Intenta arreglarlo por tu cuenta y busca ayuda en línea, no te rindas y habla acerca del problema y ofrece explicaciones de las posibles

soluciones mientras las implementas. Esto ayuda a otros a seguirte durante el proceso. Como dije antes, no tiene sentido seguir videos perfectos donde el instructor nunca resuelve problemas que surgen de improvisto.

Algunas observaciones sobre el aspecto técnico de la grabación: Revisa la configuración de audio antes de grabar un video de larga duración. Debe tener el volumen correcto y la calidad debe ser aceptable. Con respecto al editor/IDE/terminal, asegúrate de incrementar el tamaño de la fuente. Quizá sea posible posicionar el navegador web y el código lado a lado, de no ser posible, coloca ambas aplicaciones en pantalla completa y cambia entre una y otra (CMD + Tab en MacOS).

- Edita el video antes de subirlo a YouTube. No tiene que ser de alta calidad, pero trata de mantenerlo conciso para tu audiencia (por ejemplo: Omite las lecturas y resume los pasos con tus propias palabras).

Cuándo estés listo, puedes contactarme para ayudarte a promocionar cualquier material que publiques. Por ejemplo, si el video resulta bien, me encantaría incluirlo en este libro como material suplementario oficial. Solo debes contactarme una vez te sientas listo. Espero que aceptes este reto para optimizar tu experiencia de aprendizaje y ayudar a otros. Mis mejores deseos.

Involucrados

Muchas personas han hecho posible presentar *The Road to Learn React*, que es uno de los libros sobre React.js en circulación mas descargados actualmente. El autor original es el ingeniero de software Alemán Robin Wieruch[29], aunque las traducciones del libro no serían posibles sin la ayuda de muchas otras personas. Los co-autores de esta versión en Español son:

Emanuel Mathias Canova

LinkedIn: Emanuel Canova[30]

Twitter: @EmanuelCanova[31]

Instagram: @canovaEmanuel[32]

Web: www.emanuelcanova.com[33]

Argentino con pasión por desarrollar y poder contribuir a proyectos, este proyecto ha sido un hermoso camino junto a mis compañeros. Soy desarrollador BackEnd con .Net C# y de UI y UX con React + Redux. Colaborar en un proyecto es la mejor manera de contribuir al mundo de manera positiva.

Tambien soy fotógrafo por pasión e intento mejorar mis habilidades tanto como desarrollador y como fotógrafo, pero lo mas importante es mejorar como persona cada día un poco más.

Mi consejo es: “Nunca Pares de Aprender”.

Juan Luis Chaurant

LinkedIn: Juan Luis Chaurant[34]

Twitter: @juanluis_nt[35]

Web: juanluischaurant.github.io[36]

Apasionado por el desarrollo de software y el trabajo en equipo, participar en este proyecto me permitió conocer excelentes personas sin las cuáles el resultado final no sería el mismo. Soy estudiante de Ingeniería Informática, enfocado en el desarrollo de sistemas información.

[29] https://www.robinwieruch.de/
[30] https://www.linkedin.com/in/emanuel-canova
[31] https://twitter.com/EmanuelCanova
[32] https://www.instagram.com/canovaemanuel/?hl=es-la
[33] http://emanuelcanova.com/
[34] https://www.linkedin.com/in/juan-luis-chaurant-b93931139/
[35] https://twitter.com/juanluis_nt
[36] https://juanluischaurant.github.io/

También soy guitarrista y esto ha influenciado directamente mi forma de entender el mundo. Estar hoy aquí, en buena parte se debe a todos los años invertidos practicando para ser mejor músico y mejor persona.

Guillermo Salazar

Twitter: @fenomemoide[37]

Apasionado por la tecnología. Curioso por el desarrollo web, seguridad informática y productividad.

Espero esta pequeña contribución sea de ayuda para muchos. Agradezco a Robin, Juan, Emanuel y demás involucrados por hacer posible esta publicación.

[37] https://twitter.com/fenomemoide

Introducción a React

Este capítulo es una introducción a React, una librería de JavaScript para renderizar interfaces en una página única y aplicaciones móviles, donde explico por qué los desarrolladores deberían considerar agregar la librería React a sus herramientas. Nos sumergiremos en el ecosistema de React para crear una aplicación desde cero sin configuración. En el camino introduciremos **JSX**, la sintaxis para React y **ReactDOM**, para que tengas entendimiento de los usos prácticos de React en aplicaciones web modernas.

Hola, mi nombre es React.

En los últimos años las Single Page Applications o Aplicaciones de Página Única (SPA por sus siglas en Inglés[38]) se han vuelto populares. Frameworks como Angular, Ember y Backbone permiten a los desarrolladores JavaScript a construir aplicaciones web modernas más allá de lo que se podría lograr usando JavaScript puro (Vanilla JavaScript) y jQuery. Los tres frameworks mencionados están entre los primeros SPAs, apareciendo entre 2010 y 2011, pero hay muchas más opciones para el desarrollo de una sola página. La primera generación de frameworks SPA llegó en un nivel empresarial, por lo que eran menos flexibles. React, por otro lado, permanece como una librería innovadora que ha sido adoptada por líderes tecnológicos como (Airbnb, Netflix y Facebook[39]).

La versión inicial de React fue lanzada en 2013 por el equipo de desarrollo web de Facebook en 2013 como una librería para la capa de vistas, que representa la V en el Modelo Vista Controlador (MVC por sus siglas en Inglés)[40]. Esta librería permite renderizar componentes como elementos visibles en un navegador, mientras que con todo el ecosistema React, es posible crear aplicaciones de una sola página. Mientras que los primeros frameworks trataron de resolver muchos problemas a la vez, React solo es usado para construir tu capa de vista. Específicamente es una librería donde la vista es una jerarquía de componentes ensamblables(composable).

En React, el enfoque permanece en la capa de Vistas hasta que se agreguen más aspectos a la aplicación. Estos son bloques de construcción para una SPA, que son esenciales para construir una aplicación madura, y tienen dos ventajas:

*Puedes aprender a usar los bloques de construcción uno a la vez, sin preocuparte por entenderlos totalmente desde el principio. Esto es muy diferente a un Framework que incluye todo el set de bloques de construcción desde que inicias el proyecto. Este libro presenta a React como el primer bloque de construcción. Más bloques de construcción vendrán luego.

*Todos los bloques de construcción son intercambiables. Lo que hace del ecosistema React algo innovador donde múltiples soluciones compiten entre sí para solucionar un problema, así que puedes elegir la solución más atractiva para ti y tu caso de uso.

Hoy en día, React es probablemente una de las mejores opciones para la construcción de sitios web modernos. De nuevo, React sólo se encarga de la capa de vista, pero gracias a su amplio ecosistema, representa un framework completo, flexible e intercambiable[41]. Además, cuenta con una API ligera, un ecosistema impresionante y una gran comunidad.

Ejercicios

Si quieres saber más sobre por qué elegí React o información más profunda sobre los temas mencionados arriba, éstos artículos pueden darte una mayor perspectiva:

[38] https://es.wikipedia.org/wiki/Single-page_application
[39] https://github.com/facebook/react/wiki/Sites-Using-React
[40] https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador
[41] https://www.robinwieruch.de/essential-react-libraries-framework/

- lee acerca de por qué me cambié de Angular a React[42]
- lee acerca de el flexible ecosistema de React[43]
- lee acerca de cómo aprender a usar un framework[44]

[42] https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/
[43] https://www.robinwieruch.de/essential-react-libraries-framework/
[44] https://www.robinwieruch.de/how-to-learn-framework/

Requerimientos

Antes de seguir avanzando debes estar familiarizado con los fundamentos del desarrollo web. Se espera que sepas trabajar con HTML, CSS y JavaScript, además de lo que significa el término API[45], estarás usando esto durante todo el libro.

Te animo a que te unas al canal Slack[46] oficial del libro para obtener ayuda o ayudar a otros.

Editor y Terminal

Para las lecciones necesitarás un editor de texto plano o IDE y la terminal (línea de comandos). Puedes leer mi guía de configuración[47] para ayudarte a organizar tus herramientas, esta guía está pensada para usuarios de Mac, sin embargo todas las herramientas que necesitarás están disponibles para varios sistemas operativos. Opcionalmente, puedes utilizar Git y GitHub por tu cuenta mientras estés realizando los ejercicios del libro, para mantener tus proyectos en repositorios de GitHub. Aquí dispones de una pequeña guía[48] sobre cómo usar estas herramientas. Sin embargo, no es obligatorio su uso, podría resultar extenuante intentar aprenderlo todo al mismo tiempo.

Node y npm

Por último, necesitarás instalar Node y npm[49]. Ambos se utilizan para administrar las bibliotecas que necesitarás en El Camino para aprender React. Instalarás paquetes externos Node a través del Gestor de Paquetes Node (npm, por sus siglas en Inglés). Estos paquetes pueden constituir bibliotecas o frameworks completos.

Puedes verificar la versiones instaladas Node y npm, respectivamente dentro de la terminal. Si no obtienes ninguna salida en la terminal, así que debes verificar tu instalación Node y npm. Estas son mis versiones al momento de escribir este libro:

Command Line

```
node --version
*v10.11.0
npm --version
*v6.4.1
```

A continuación, un pequeño curso intensivo Node y npm. No es exhaustivo, pero obtendrás todas las herramientas necesarias. Si ya estás familiarizado con estas puedes omitir este capítulo.

[45] https://www.robinwieruch.de/what-is-an-api-javascript/
[46] https://slack-the-road-to-learn-react.wieruch.com/
[47] https://www.robinwieruch.de/developer-setup/
[48] https://www.robinwieruch.de/git-essential-commands/
[49] https://nodejs.org/es/

El gestor **npm** permite instalar **paquetes externos** desde la terminal. Estos paquetes pueden ser un conjunto de funciones de utilidad, bibliotecas o frameworks enteros. Estos representan dependencias de tu aplicación, y puedes instalarlos en tu carpeta global de paquetes Node o bien en la carpeta local de tu proyecto.

Los paquetes globales Node son accesibles desde cualquier lugar de la terminal y sólo hay que instalarlos una vez en el directorio global. Puedes instalar un paquete a nivel global escribiendo en la terminal:

Command Line

```
npm install -g <paquete>
```

La etiqueta `-g` indica a npm que debe instalar el paquete a nivel global. Los paquetes locales son utilizados en tu aplicación. Por ejemplo, React como una librería será un paquete local, y puede ser requerido en tu aplicación para su uso. Puedes instalarlo a través de la terminal escribiendo:

Command Line

```
npm install react
```

El paquete instalado aparecerá automáticamente en una carpeta llamada *node_modules/* y será agregado al archivo *package.json* junto a las otras dependencias de tu aplicación.

Ahora, ¿cómo inicializar la carpeta *node_modules/* y el archivo *package.json* en tu proyecto? Para ello existe un comando que inicia un proyecto npm que incluye automáticamente un archivo *package.json.* Sólo cuando tu proyecto posee este archivo, puedes instalar nuevos paquetes locales vía npm.

Command Line

```
npm init -y
```

La etiqueta `-y` es un acceso directo para inicializar todos los valores predeterminados en el archivo *package.json* correspondiente. Inmediatamente, con npm inicializado en tu carpeta de proyecto, estás listo para instalar nuevos paquetes vía `npm install <paquete>`.

Un dato extra sobre el archivo *package.json.* Este archivo permite que compartas tu proyecto con otros desarrolladores sin compartir todos los paquetes Node. Contiene todas las referencias de los paquetes Node utilizados en tu proyecto. Y estos paquetes son llamados dependencias. Cualquiera puede copiar tu proyecto sin las dependencias, pues, estas son referenciadas en el archivo *package.json.* Así, cuando alguien copia tu proyecto, puede instalar todos las dependencias usando `npm install` en la terminal.

Hay otro comando npm que quiero mencionar, para prevenir confusiones:

Command Line

```
npm install --save-dev <paquete>
```

La etiqueta `--save-dev` indica que el paquete Node es sólo usado en el entorno de desarrollo, esto quiere decir que no será usado en producción cuando cuelgues tu aplicación en un servidor. Esto es útil para realizar pruebas a una aplicación usando un paquete Node, quieres excluirlo de tu entorno de producción.

Podrías querer usar otros gestores para trabajar con los paquetes Node. **Yarn** es un gestor de dependencias que trabaja de forma similar a **npm**. Tiene su propia lista de instrucciones, aún así tienes acceso al mismo registro de npm. Yarn fue creado para resolver problemas que npm no pudo, pero ambas herramientas han evolucionado hasta el punto de que cualquiera de las dos sería suficiente.

Ejercicios:

- configura un proyecto npm
  - crea una nueva carpeta con `mkdir <nombre_de_la_carpeta>`
  - navega hasta la carpeta con `cd <nombre_de_la_carpeta>`
  - ejecuta `npm init -y`
  - instala un paquete local, como React, con `npm install --save react`
  - échale un vistazo al archivo *package.json* y a la carpeta *node_modules/*
  - descubre cómo desinstalar el paquete Node *react*
- lee más sobre npm[50]

[50] https://docs.npmjs.com/

Instalación

Existen varias maneras de comenzar una aplicación React. La primera es utilizar una Content Delivery Network[51], o Red de Entrega de Contenidos (CDN por sus siglas en Inglés). Esto puede sonar complicado, pero no lo es. Muchas compañías usan CDNs que almacenan públicamente archivos para sus usuarios. Algunos de estos archivos que pueden ser una librería como React, ya que la librería empaquetada de React es únicamente un archivo JavaScript *react.js*.

Para utilizar React usando una CDN, puedes hacerlo insertando la etiqueta `<script>` en tu código HTML, con la URL de la CDN que desees utilizar. Para React necesitas dos archivos (librerías): *react* y *react-dom*.

Code Playground

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></scri\
pt>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js\
"></script>
```

También puedes incluir React en tu aplicación inicializándola como un proyecto Node. Cuando tu aplicación tiene un archivo *package.json* es posible instalar *react* y *react-dom* desde la terminal. El único requisito es que la carpeta esté inicializada como un proyecto npm, lo puedes hacer utilizando `npm init -y` tal como se mostró anteriormente. Es posible instalar múltiples paquetes en una misma línea a la vez con npm.

Command Line

```
npm install react react-dom
```

El enfoque anterior se utiliza a menudo para añadir React a una aplicación existente administrada con npm.

Puede que tengas que lidiar con Babel[52] también para hacer tu aplicación compatible con JSX (la sintáxis de React) y JavaScript ES6. Babel transpila tu código–es decir, lo convierte a vanilla JavaScript– para que los navegadores puedan interpretar ES6 y JSX. Debido a la dificultad de esta tarea, Facebook desarrolló *create-react-app*, como una solución *cero-configuraciones* de React. La siguiente sección muestra cómo iniciar con el desarrollo de tu aplicación con esta herramienta.

Ejercicios:

- lee más acerca de cómo instalar React[53]

[51] https://es.wikipedia.org/wiki/Red_de_entrega_de_contenidos
[52] http://babeljs.io/
[53] https://facebook.github.io/react/docs/installation.html

Cero Configuraciones

En El Camino para aprender React usarás create-react-app[54] para iniciar el desarrollo tu aplicación. Este kit lanzado por Facebook en 2016, permite rápidamente empezar a trabajar en tu aplicación sin preocuparte por configuraciones. La gente lo recomienda a principiantes en un 96%[55]. Cuando utilizas *create-react-app* las herramientas y configuración evolucionan en segundo plano, mientras que el foco se mantiene en la implementación de la aplicación.

Para empezar, deberás agregar el *create-react-app* a tus paquetes globales Node. A partir de ahora estará disponible en la terminal para inicializar nuevas aplicaciones React.

Command Line

```
npm install -g create-react-app
```

En la terminal puedes comprobar la versión de *create-react-app* para verificar que la instalación fue exitosa:

Command Line

```
create-react-app --version
*v2.0.2
```

Ahora, puedes comenzar con el desarrollo de tu primera aplicación React. La llamaremos *hacker-news*, puedes escoger un nombre distinto. Iniciar tu aplicación tomará unos pocos segundos. Después de esto, simplemente navega hasta el nuevo directorio con tu terminal:

Command Line

```
create-react-app hackernews
cd hackernews
```

Ya puedes abrir la aplicación en tu editor de texto. La siguiente estructura de carpetas o variación de esta depende de la versión de *create-react-app* que tengas instalada, deberías poder ver algo cómo esto:

[54] https://github.com/facebookincubator/create-react-app
[55] https://twitter.com/dan_abramov/status/806985854099062785

Folder Structure

```
hackernews/
  README.md
  node_modules/
  package.json
  .gitignore
  public/
    favicon.ico
    index.html
    manifest.json
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    registerServiceWorker.js
```

A continuación, una pequeña descripción de cada una de las carpetas y archivos que encontrarás dentro del directorio recién creado. Está bien si no entiendes todo al principio.

- **README.md:** La extensión .md indica que el archivo está en formato "markdown". Markdown es un lenguaje de marcado ligero con sintaxis de texto plano. Muchos códigos fuente de proyectos incluyen un archivo *README.md* con instrucciones acerca del proyecto. Cuando estés subiendo tu proyecto a una plataforma como GitHub, eventualmente el archivo *README.md* mostrará promisoriamente su contenido cuando alguien acceda al repositorio. Como utilizaste *create-react-app*, tu archivo *README.md* debería ser igual al que se puede observar en el repositorio GitHub de create-react-app[56] oficial.
- **node_modules/**: Contiene todos los paquetes Node que han sido instalados vía npm. Como utilizaste *create-react-app* para inicializar tu aplicación, debes de poder ver un par de módulos Node ya instalados. Usualmente nunca tocarás esta carpeta, pues, con npm puedes instalar y desinstalar paquetes Node desde la terminal.
- **package.json**: Muestra una lista de las dependencias de paquetes Node y un conjunto de otras configuraciones referentes al proyecto.
- **.gitignore**: Especifica los archivos y carpetas que no serán añadidos a tu repositorio de git. Archivos que sólo vivirán en el proyecto local.
- **public/**: Almacena todos los archivos raíz de desarrollo, como por ejemplo *public/index.html*, este índice es el que se muestra en la dirección localhost:3000 cuando estás desarrollando tu aplicación. *create-react-app* viene ya configurado para relacionar este archivo índice con todos los scripts en la carpeta *src/*.

[56] https://github.com/facebookincubator/create-react-app

- **build/**: Esta carpeta será creada cuando el proyecto se esté preparando para la etapa de producción y contendrá todos los archivos relacionados a esta etapa de desarrollo. Todo el código escrito en las carpetas *src/* y *public/* serán empaquetados en un par de archivos y posicionados en la carpeta *build/* cuando el proyecto se esté compilando.
- **manifest.json** y **registerServiceWorker.js**: Por el momento no te preocupes acerca de lo que estos archivos hacen, no los necesitaremos en este proyecto.

No necesitas tocar los archivos que acabamos de mencionar. Por ahora todo lo que necesitas está localizado en la carpeta *src/* y la mayor atención recae sobre el archivo *src/App.js*, que utilizaremos para implementar componentes React.

Más adelante, podrás dividir los componentes React en múltiples archivos, con uno o más componentes dentro de cada archivo.

Adicionalmente, encontrarás un archivo *src/App.test.js* para pruebas y uno *src/index.js* como punto de entrada al mundo React. Explorarás ambos archivos en capítulos próximos. También existe un archivo *src/index.css* y un archivo *src/App.css* que sirven para darle estilos a tu aplicación y sus componentes. Estos ya incluyen algunos estilos por defecto.

La aplicación *create-react-app* es un proyecto npm, puedes utilizar npm para instalar y desinstalar paquetes Node en tu proyecto, adicionalmente, incluye algunos scripts que puedes ejecutar desde la terminal:

Command Line

```
# Ejecuta la aplicación en http://localhost:3000
npm start

# Ejecuta las pruebas
npm test

# Prepara la aplicación para la etapa de construcción
npm run build
```

Estos scripts se encuentran definidos en el archivo *package.json*. Tu aplicación React se encuentra ya inicializada, y lo mejor está por venir, cuándo ejecutes tu aplicación en el navegador.

Ejercicios:

- ejecuta en la terminal, el comando `npm start` y visita la aplicación en tu navegador (puedes cancelar la ejecución de este comando presionando Control + C)
- ejecuta el comando interactivo `npm test`
- ejecuta el comando `npm run build` y verifica que la carpeta *build/* sea añadida a tu proyecto (puedes removerla después; nota que la carpeta build puede ser usada más tarde para colocar tu aplicación en línea[57])

[57] https://www.robinwieruch.de/deploy-applications-digital-ocean/

- familiarízate con la estructura de carpetas
- familiarízate con el contenido de los archivos
- lee más sobre los comandos npm y create-react-app[58]

[58] https://github.com/facebookincubator/create-react-app

Introducción a JSX

Ahora conocerás JSX, la sintaxis empleada por React. Como fue mencionado antes, *create-react-app* ya ha iniciado una aplicación estándar. Todos los archivos incluyen implementaciones predeterminadas. Es tiempo de sumergirnos en el código fuente.

El único archivo que manipularás por ahora es: *src/App.js*.

src/App.js

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

No te dejes intimidar por las instrucciones import/export y la declaración de clases. Estas características pertenecen a JavaScript ES6, volveremos a ellas más adelante.

Dentro del archivo *src/App.js* se encuentra un **React ES6 class component o componente de clase React ES6** con el nombre `App`. Esto es una declaración de componente.

Básicamente, después de haber declarado un componente puedes utilizarlo como elemento en cualquier parte de tu aplicación produciendo una **instancia** de tu **componente**, o mejor dicho: Instanciando el componente.

El **elemento** que es devuelto se especifica dentro del método `render()`. Los elementos son de lo que están hechos los componentes. Es útil que entiendas las diferencias entre los términos “componente”, “instancia” y “elemento”.

En un momento verás donde se instancia el componente `App` y cómo se renderiza en el navegador.

El componente `App` es sólo la declaración y por si solo no hará nada. Para utilizar dicho componente podrías instanciarlo en algún lugar de tu JSX con la etiqueta `<App />`.

El contenido dentro del bloque de código perteneciente al método `render()` parece ser HTML, pero en realidad es JSX, y te permite mezclar HTML y JavaScript. Es potente, pero confuso cuando estás acostumbrado a HTML simple. Por eso que un buen punto de partida es comenzar utilizando HTML básico en tu JSX. Las expresiones JavaScript las puedes utilizar insertándolas entre corchetes.

Primero, vamos a eliminar todo el contenido por defecto que es retornado por `render` y agregar nuestro propio HTML.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <h2>Bienvenido al Camino para aprender React</h2>
      </div>
    );
  }
}

export default App;
```

Ahora, el método `render()` sólo devuelve HTML sin JavaScript. Vamos a definir "Bienvenido al Camino para aprender React" como el valor de una variable, que puede ser utilizad en JSX usando corchetes.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    var helloWorld = 'Bienvenido al Camino para aprender React';
    return (
      <div className="App">
        <h2>{helloWorld}</h2>
      </div>
```

```
    );
  }
}

export default App;
```

Cuando inicies tu aplicación desde la terminal con `npm start` todo debería funcionar.

Además, probablemente notaste el atributo `className`. Este es similar al atributo estándar `class` en HTML. Debido a razones técnicas, JSX tuvo que reemplazar varios atributos internos de HTML. Puedes ver todos los atributos HTML compatibles con JSX en la documentación de React[59] aquí. Más adelante conoceremos nuevos atributos JSX.

Ejercicios:

- define más variables y renderízalas en tu JSX
  - utiliza un complex object u objeto complejo en español, para representar a un usuario con nombre y apellido
  - renderiza las propiedades del usuario utilizando JSX
- lee más sobre JSX[60]
- lee más sobre componentes, elementos e instancias en React[61]

[59] https://facebook.github.io/react/docs/dom-elements.html
[60] https://facebook.github.io/react/docs/introducing-jsx.html
[61] https://facebook.github.io/react/blog/2015/12/18/react-components-elements-and-instances.html

ES6 const y let

Probablemente notaste que declaramos la variable `helloWorld` con `var`. JavaScript ES6 incluye otras dos formas de declarar variables: `const` y `let`. En JavaScript ES6 rara vez utilizarás `var`. Una variable declarada con `const` no se puede volver a asignar o volver a declarar. No puede ser mutada (cambiada o modificada), es decir, la estructura de datos se vuelve inmutable. Una vez que se define la estructura de datos, no se puede cambiar.

Code Playground

```
// no permitido
const helloWorld = 'Bienvenido al Camino para aprender React';
helloWorld = 'Bye Bye React';
```

Por otra parte, una variable declarada con `let` puede mutar.

Code Playground

```
// permitido
let helloWorld = 'Bienvenido al Camino para aprender React';
helloWorld = 'Hasta luego, React';
```

TIP Declara variables usando `let` cuando pienses reasignarlas.

Nota que la variable declarada con `const` no se puede modificar. Pero cuando el valor de la variable es un arreglo o un objeto, sus valores pueden ser alterados. Es decir, los valores dentro del objeto o arreglo no son inmutables.

Code Playground

```
// permitido
const helloWorld = {
  text: 'Bienvenido al Camino para aprender React'
};
helloWorld.text = 'Hasta luego, React';
```

Hay diferentes opiniones sobre cuándo usar *const* y *let*. Sugiero usar `const` siempre que sea posible, indicando así que se desea mantener la estructura de datos inmutable aunque los valores en objetos y arreglos se puedan modificar. La inmutabilidad es ampliamente adoptada en React y su ecosistema. Es por eso que `const` debe ser la opción por defecto cuando se define una variable, aunque no se trata de inmutabilidad realmente, sino de asignar las variables una sola vez. Esto muestra la intención de no cambiar(reasignar) la variable incluso cuando el su contenido puede ser modificado.

En tu aplicación utiliza `const` en vez de `var`.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    const helloWorld = 'Bienvenido al Camino para aprender React';
    return (
      <div className="App">
        <h2>{helloWorld}</h2>
      </div>
    );
  }
}

export default App;
```

Ejercicios:

- lee más sobre ES6 const[62]
- lee más sobre ES6 let[63]
- investiga más acerca de las estructuras de datos inmutables
  - ¿Por qué se utilizan en la programación en general?
  - ¿Por qué se utilizan en React?

[62] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/const
[63] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/let

ReactDOM

Antes de volver a trabajar en el componente `App` es posible que quieras ver dónde es utilizado. Este se encuentra declarado dentro del archivo *src/index.js*.

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Básicamente `ReactDOM.render()` usa un nodo DOM perteneciente al HTML de tu aplicación para reemplazarlo con JSX. Así es como puedes integrar fácilmente React a cualquier aplicación externa. No está prohibido utilizar `ReactDOM.render()` varias veces en una aplicación. Puedes utilizarlo en varios lugares para iniciar la sintaxis JSX simple, un componente React, múltiples componentes React o una aplicación completa.

`ReactDOM.render()` por defecto requiere dos argumentos:

El primero es JSX que será renderizado.

Y el segundo especifica el lugar en el que la aplicación React será insertada dentro del código HTML de tu aplicación. En este ejemplo, se selecciona un elemento con `id='root'`. Puedes abrir el archivo *public/index.html* y encontrar allí dicho elemento.

En el ejercicio, `ReactDOM.render()` ya incluye el componente `App` como primer parámetro. Sin embargo, en el siguiente ejemplo utilizaremos código JSX simple y no la instancia de un componente. Todo esto para demostrar que al utilizar JSX simple no tienes es necesario instanciar un componente.

Code Playground

```
ReactDOM.render(
  <h1>Hola Mundo de React</h1>,
  document.getElementById('root')
);
```

Ejercicios:

- abre *public/index.html* para ver dónde se conectan las aplicaciones React con el HTML de tu aplicación

- lee más sobre cómo renderizar elementos[64]

[64] https://facebook.github.io/react/docs/rendering-elements.html

Reemplazo de Módulos en Caliente

Hot Module Replacement o Reemplazo de Módulos en Caliente (HMR por sus siglas en Inglés) es una interesante característica que puedes implementar en el archivo *src/index.js* para mejorar tu experiencia como desarrollador. Pero es opcional y puede que resulte abrumador al principio.

En *create-react-app* ya es una ventaja que el navegador recargue automáticamente la página tan pronto el código fuente cambia. Inténtalo, cambia la variable `helloWorld` en tu archivo *src/App.js*. El navegador debe recargar la página automáticamente. Sin embargo, esto puede hacerse de una mejor manera.

El Reemplazo de Módulo Caliente es una herramienta para actualizar tu aplicación dentro del navegador. Es decir, el navegador no actualiza la página entera. En *create-react-app* esta herramienta puede ser fácilmente activada, sólo necesitas agregar unas pocas líneas de código en tu *src/index.js* - tu punto de entrada de React -.

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';

ReactDOM.render(
  <App />,
  document.getElementById('root')
);

if (module.hot) {
  module.hot.accept();
}
```

Eso es todo. Ahora, intenta cambiar de nuevo la variable `hellowWorld` en tu archivo *src/App.js*. El navegador no recargará la página, sin embargo, notarás que la aplicación vuelve a cargar y muestra la salida correcta.

HMR ofrece múltiples ventajas:

Imagina que estás depurando tu código con declaraciones `console.log()`. Estas declaraciones permanecerán en la consola de desarrolladores, incluso si el código cambia, pues el navegador ya no actualiza la página. Lo que puede resultar conveniente al momento de depurar.

En una aplicación en crecimiento, una actualización de página retrasa la productividad. Tienes que esperar hasta que la página se cargue, y una recarga de página puede tardar varios segundos en una aplicación grande. HMR soluciona este inconveniente.

Sin embargo, el mayor beneficio que ofrece HMR es que puede mantener el estado de la aplicación. Imagina que tienes un diálogo en tu aplicación que consta de varios pasos y estás en el paso 3. Sin HMR el código fuente cambiaría y el navegador actualizaría la página, después tendrías que abrir el cuadro de diálogo nuevamente y navegar desde el paso 1 hasta el paso 3.

Con HMR el diálogo permanece abierto en el paso 3. Es decir, mantiene el estado de la aplicación, aunque el código fuente cambie. La aplicación misma vuelve a cargar, más no la página.

Ejercicios:

- cambia el código fuente de *src/App.js* varias veces para ver a HMR en acción
- mira los primeros 10 minutos de Live React: Hot Reloading with Time Travel[65] por Dan Abramov

[65] https://www.youtube.com/watch?v=xsSnOQynTHs

JavaScript avanzado en JSX

Volvamos a trabajar en el componente `App`. Hasta ahora logramos renderizar algunas variables primitivas con JSX. Ahora, comenzarás a renderizar una lista de artículos. La lista en principio contendrá datos artificiales, pero más adelante recibirá los datos desde una API externa, lo que será mucho más emocionante.

Primero, define la lista de elementos.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

const list = [
  {
    title: 'React',
    url: 'https://facebook.github.io/react/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  },
  {
    title: 'Redux',
    url: 'https://github.com/reactjs/redux',
    author: 'Dan Abramov, Andrew Clark',
    num_comments: 2,
    points: 5,
    objectID: 1,
  },
];

class App extends Component {
  ...
}
```

Los datos artificiales representan los datos que más adelante serán extraídos de la API. Cada elemento dentro de la lista tiene un título, una URL y un autor. Además incluye un identificador, puntos (que indican la popularidad de un artículo) y un recuento de comentarios.

Ahora, puedes utilizar el método `map` de JavaScript[66] incorporándolo dentro del código JSX. El método `map` permite iterar sobre la lista de elementos para poder mostrarlos. Recuerda que dentro del código JSX debes encerrar entre corchetes la expresión JavaScript:

[66] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/map

src/App.js

```
class App extends Component {
  render() {
    return (
      <div className="App">
        {list.map(function(item) {
          return <div>{item.title}</div>;
        })}
      </div>
    );
  }
}

export default App;
```

Usar JavaScript junto con HTML en JSX es muy poderoso. Es posible que antes hayas utilizado el método `map` para convertir una lista de elementos a otra, pero esta vez lo utilizas para convertir una lista de elementos a elementos HTML.

Code Playground

```
const array = [1, 4, 9, 16];

// pasa una función a map
const newArray = array.map(function (x) { return x * 2; });

console.log(newArray);
// resultado esperado: Array [2, 8, 18, 32]
```

Por ahora sólo se muestra el valor de la propiedad `title` correspondiente a cada elemento de la lista. A continuación, vamos a mostrar otras propiedades del artículo.

src/App.js

```
class App extends Component {
  render() {
    return (
      <div className="App">
        {list.map(function(item) {
          return (
            <div>
              <span>
                <a href={item.url}>{item.title}</a>
```

```
              </span>
              <span>{item.author}</span>
              <span>{item.num_comments}</span>
              <span>{item.points}</span>
            </div>
          );
        })}
      </div>
    );
  }
}

export default App;
```

Puedes ver como el método `map` está sangrado dentro del código JSX. Cada propiedad de elemento se muestra en una etiqueta `<span>`. Además, la propiedad URL del elemento se utiliza en el atributo `href` de la etiqueta de anclaje.

React hará todo el trabajo por ti y mostrará cada elemento. Aunque puedes hacer algo para ayudar a que React alcance su máximo potencial y tenga un mejor rendimiento. Al asignar un atributo clave a cada elemento de lista, React será capaz de identificar cada elemento de la lista, React puede identificar elementos cambiados y eliminados cuando la lista cambie. Esta lista de ejemplo tiene un identificador:

src/App.js

```
{list.map(function(item) {
  return (
    <div key={item.objectID}>
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
    </div>
  );
})}
```

Asegúrate de que el atributo clave tiene un valor estable. No cometas el error de usar el índice asignado para el elemento dentro del arreglo. El índice del arreglo no es del todo estable. Por ejemplo, cuando el orden de la lista cambie, React tendrá dificultades para identificar los elementos correctamente, pues cada elemento dentro de la lista tendrá ahora un orden y un índice distinto.

src/App.js

```
// No hagas esto
{list.map(function(item, key) {
  return (
    <div key={key}>
      ...
    </div>
  );
})}
```

Ahora puedes iniciar tu aplicación desde la terminal, abrir tu navegador y ver los dos elementos desplegados correspondientes a la lista que acabas de agregar al código de tu componente.

Ejercicios:

- lee más sobre llaves y listas de elementos en React[67]
- repasar funcionalidades estándar para Arreglos en JavaScript[68]
- utiliza más expresiones JavaScript por tu cuenta en JSX

[67] https://facebook.github.io/react/docs/lists-and-keys.html
[68] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/map

Funciones Flecha en ES6 (Arrow Functions)

JavaScript ES6 introdujo las funciones flecha, que resultan más cortas que expresiones de función tradicionales.

Code Playground

```
// declaración de una función
function () { ... }

// declaración de una función flecha
() => { ... }
```

Puedes quitarlos cuando la función sólo recibe un argumento, pero tienes que conservarlos cuando hay múltiples argumentos.

Code Playground

```
// permitido
item => { ... }

// permitido
(item) => { ... }

// no permitido
item, key => { ... }

// permitido
(item, key) => { ... }
```

Ahora, revisemos nuevamente el método `map`. Puedes declararlo de manera más concisa con una función flecha ES6.

src/App.js

```
{list.map(item => {
  return (
    <div key={item.objectID}>
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
```

```
    </div>
  );
})}
```

Además, es válido eliminar el *cuerpo del bloque* de la función flecha ES6. En un *cuerpo conciso* un `return` implícito se adjunta, de modo que se puede quitar la declaración `return`. Esto sucederá a menudo en el libro, así que asegúrate de entender la diferencia entre un cuerpo de bloque y un cuerpo conciso dentro de las funciones flecha ES6.

src/App.js

```
{list.map(item =>
  <div key={item.objectID}>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
  </div>
)}
```

Ahora tu código JSX es más conciso y legible. Al declarar `map` de esta manera, se omite la sentencia "function", los corchetes y la declaración `return`.

Ejercicios:

- lee más sobre funciones flecha ES6[69]

[69] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Funciones/Arrow_functions

Clases ES6 (ES6 Classes)

JavaScript ES6 introdujo el uso clases, que comúnmente son utilizadas en lenguajes de programación orientados a objetos. JavaScript fue y sigue siendo muy flexible en sus paradigmas de programación. Funciona bien tanto con programación funcional como con programación orientada a objetos, lado a lado, para sus casos de uso particulares.

React adopta el paradigma programación funcional, sin embargo, al crear estructuras de datos inmutables las clases se utilizan para declarar componentes y se les llama componentes de clase ES6. React aprovecha las mejores cualidades de ambos paradigmas de programación.

Considera la siguiente clase llamada `Developer` para examinar una clase de JavaScript ES6 sin pensar en componentes React.

Code Playground

```
class Developer {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  getName() {
    return this.firstname + ' ' + this.lastname;
  }
}
```

Una clase tiene un constructor que permite instanciarla. Este constructor puede tomar argumentos y asignarlos a la instancia de la clase. Además, una clase puede definir funciones, y dado que la función está asociada a una clase, se le llama método de clase.

En el ejemplo anterior, `class Developer` es sólo la declaración de la clase. Es posible crear varias instancias de una clase por medio de la invocación. Lo que resulta similar al componente de clase ES6, que tiene una declaración pero debe ser usado en otro lugar para instanciarlo:

Code Playground

```
const robin = new Developer('Robin', 'Wieruch');
console.log(robin.getName());
// output: Robin Wieruch
```

React utiliza clases JavaScript ES6 en los componentes de clase ES6. Ya utilizaste un componente de clase ES6 anteriormente.

src/App.js

```
import React, { Component } from 'react';

...

class App extends Component {
  render() {
    ...
  }
}
```

La clase `App` se extiende desde otro `Component`. En la programación orientada a objetos se emplea el principio de herencia, que hace posible pasar funcionalidades de una clase a otra clase.

La clase `App` extiende la funcionalidad de la clase `Component`. Para ser más específicos, `App` hereda funcionalidades de la clase Component. Este componente se utiliza para extender una clase ES6 básica a una clase de componente ES6. Tiene todas las funcionalidades que un componente necesita. Y una de estas funcionalidades es un método que ya conoces, el método `render()` del que conocerás más funcionalidades más adelante.

La clase `Component` encapsula todas las funcionalidades de React que un desarrollador no necesita ver. Permite a los desarrolladores utilizar las clases como componentes en React.

Los métodos encapsulados dentro de la clase `Component` representan la interfaz pública. Uno de estos métodos debe ser sobrescrito, para los demás no es necesario. Aprenderás acerca de estos últimos cuando el libro llegue a los métodos del ciclo de vida en un capítulo próximo. El método `render()` tiene que ser sobrescrito, porque define la salida de React `Component`.

Ahora que ya conoces los conceptos básicos de las clases JavaScript ES6 y cómo se utilizan en React para extenderlas a componentes, aprenderás más sobre los métodos de Componente cuando el libro describa los Métodos de Ciclo de Vida de React.

Ejercicios:

- lee sobre Fundamentos de JavaScript antes de aprender React[70]
- lee más sobre Clases en ES6[71]

¡Ahora sabes cómo arrancar tu propia aplicación React! Repasemos brevemente los visto en los últimos capítulos:

- React

[70] https://www.robinwieruch.de/javascript-fundamentals-react-requirements/
[71] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes

  - *create-react-app* arranca una aplicación React
  - JSX mezcla HTML y JavaScript para definir los componentes React
  - Componentes, instancias y elementos son cosas diferentes
  - `ReactDOM.render()` es un punto de entrada para renderizar componentes React en el DOM
  - funciones JavaScript incorporadas pueden ser utilizadas en JSX
  - El método `map` puede utilizarse para renderizar una lista de elementos como elementos HTML
- ES6
  - Declaraciones de variables con `const` y `let` pueden ser utilizadas en casos de uso particulares
  - las funciones flecha pueden utilizarse para acortar las declaraciones de funciones
  - las clases se utilizan para definir componentes en React

Es recomendable que te detengas en este punto. Internaliza lo aprendido y aplícalo por cuenta propia. Puedes experimentar con el código fuente que has escrito hasta ahora.

El código fuente está disponible en el repositorio oficial[72].

[72] https://github.com/rwieruch/hackernews-client/tree/0c5a701170dcc72fe68bdd594df3a6522f58fbb3

Conceptos Básicos en React

Este capítulo te guiará a través de los aspectos básicos de React. Conocerás lo que es el estado y las interacciones dentro de componentes. Además, verás distintas maneras de declarar un componente y cómo hacerlos reutilizables. Prepárate para darle vida propia a tus componentes React.

Estado local de un Componente

El estado local de un componente, también conocido como estado interno, te permite almacenar, modificar y eliminar propiedades almacenadas dentro de un componente. El componente de clase ES6 puede utilizar un constructor para inicializar el estado local del componente. El constructor se llama una sola vez al inicializar el componente.

Veamos el constructor de clase.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);
  }

  ...

}
```

El componente `App` es una subclase de `Component`, a esto se debe el `extends Component` en la declaración del componente `App`.

Es obligatorio llamar a `super(props);`, pues habilita `this.props` dentro de tu constructor en caso de que quieras acceder a éstas. De lo contrario, al intentar acceder a `this.props`dentro de tu constructor retornará `undefined`. En este caso, el estado inicial dentro del componente `App` debería ser la lista de elementos.

src/App.js

```
const list = [
  {
    title: 'React',
    url: 'https://reactjs.org/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  },
  ...
];

class App extends Component {
```

```
  constructor(props) {
    super(props);

    this.state = {
      list: list,
    };
  }

  ...

}
```

El estado está ligado a la clase por medio del objeto `this`, por tanto se puede acceder al estado local de todo el componente. Por ejemplo, puedes usar el estado dentro del método `render()`. Anteriormente mapeaste una lista estática de elementos dentro del método `render()` que fue definida fuera del componente `App`. Ahora, accederás a la lista almacenada en el estado local dentro del componente.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
          </div>
        )}
      </div>
    );
  }
}
```

Esta lista es ahora parte del componente, es decir se encuentra almacenada en el estado local del componente. Podrías fácilmente agregar artículos, cambiarlos o quitarlos de la lista. Cada vez que

el estado del componente cambie, el método `render()` de tu componente se ejecutará de nuevo. Así es como puedes fácilmente cambiar el estado local de un componente y asegurarte de que el componente se vuelva a renderizar y muestre la información correcta proveniente del estado local.

Ten cuidado de no mutar el estado directamente. En lugar de eso, debes utilizar un método llamado `setState()`, que conocerás a detalle en el próximo capítulo.

Ejercicios:

- experimenta con el estado local
  - define más estados iniciales dentro del constructor
  - usa y accede al estado dentro de tu método `render()`
- lee más sobre el constructor de clase ES6[73]

[73] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes

Inicializador de Objetos ES6 (ES6 Object Initializer)

En JavaScript ES6 puedes utilizar sintaxis abreviada para inicializar objetos de manera concisa. Imagina la siguiente inicialización de objeto:

Code Playground

```
const name = 'Robin';

const user = {
  name: name,
};
```

Cuando el nombre de una propiedad dentro de un objeto es igual al nombre de la variable, puedes hacer lo siguiente:

Code Playground

```
const name = 'Robin';

const user = {
  name,
};
```

En tu aplicación puedes hacer lo mismo. El nombre de la variable `list` y el nombre de la propiedad de estado comparten el mismo nombre.

Code Playground

```
// ES5
this.state = {
  list: list,
};

// ES6
this.state = {
  list,
};
```

Los nombres de método abreviados también te ayudarán mucho. En JavaScript ES6 puedes inicializar métodos dentro de un objeto de manera concisa también.

Code Playground

```
// ES5
var userService = {
  getUserName: function(user) {
    return user.firstname + ' ' + user.lastname;
  },
};

// ES6
const userService = {
  getUserName(user) {
    return user.firstname + ' ' + user.lastname;
  },
};
```

Y por último, puedes utilizar nombres de las propiedades calculados en JavaScript ES6:

Code Playground

```
// ES5
var user = {
  name: 'Robin',
};

// ES6
const key = 'name';
const user = {
  [key]: 'Robin',
};
```

Más adelante podrás usar los nombres de las propiedades computados para insertar valores dentro de un objeto de manera dinámica, una forma sencilla de generar tablas de búsqueda en JavaScript.

Ejercicios:

- experimenta con el inicializador de objetos ES6
- lee más sobre el inicializador de objetos ES6[74]

[74] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Object_initializer

Flujo de Datos Unidireccional

Ahora ya hay un estado local inicializado dentro del componente `App`. Sin embargo aún no manipulas dicho estado local. Este estado es aún estático y por lo tanto también lo es el componente. Una buena manera de experimentar con la manipulación de estado es generando interacciones entre componentes.

Agreguemos un botón a cada elemento de la lista presentada a continuación para practicar este concepto. El botón tendrá el nombre "Dismiss" y permitirá remover al elemento de la lista que lo contiene. En un cliente de correo electrónico, por ejemplo, sería útil marcar de alguna forma varios elementos de la lista como 'leídos' mientras mantienes los no leídos separados.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <button
                onClick={() => this.onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

El método de clase `onDismiss()` aún no está definido, lo definiremos en un momento. Por ahora centrémonos en el selector `onClick` del elemento `button`.

Como puedes ver, el método `onDismiss()` de la función `onClick` está encerrado dentro de otra función flecha. De esta manera, puedes ubicarte en la propiedad `objectID` perteneciente al objeto `item`, y así identificar el elemento que será eliminado al presionar el botón correspondiente. Una manera alternativa sería definiendo la función fuera de `onClick`, e incluir solamente la función definida dentro del selector. Más adelante explicaré el tema de los selectores de elementos con más detalle.

¿Notaste las multilíneas y el sangrado en el elemento `button`? Elementos con múltiples atributos en una sola línea pueden ser difíciles de leer. Es por eso que para definir el elemento `button` y sus propiedades utilizo multilíneas y sangrado, manteniendo así todo legible. Mientras que esta no es una práctica específica del desarrollo con React, es un estilo de programación recomendado para claridad y tu propia tranquilidad mental.

Ahora, tienes que implementar la función `onDismiss()`. Se necesita un `id` para identificar el elemento que se quiere eliminar. La función está vinculada a la clase `App` y por lo tanto se convierte en un método de clase, por esta razón se debe acceder a él con `this.onDismiss()` y no simplemente `onDismiss()`. El objeto `this` representa la relación de la clase. Ahora, para definir `onDismiss()` como método de clase necesitas enlazarlo con el constructor.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onDismiss = this.onDismiss.bind(this);
  }

  render() {
    ...
  }
}
```

Para el siguiente paso definimos su funcionalidad, la lógica, dentro de la clase:

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onDismiss = this.onDismiss.bind(this);
  }

  onDismiss(id) {
    ...
  }

  render() {
    ...
  }
}
```

Ahora define lo que sucede dentro del método de clase. Básicamente, quieres eliminar de la lista un artículo identificado por medio de un `id` y actualizar la lista en el estado local del componente. Al final, la lista actualizada será usada dentro del método `render()` y se mostrará en pantalla sin el elemento que recién eliminaste.

Puedes remover un elemento de una lista utilizando el método incorporado de JavaScript filter[75] que crea una lista que contiene todos los elementos de la lista original que pasan la prueba establecida.

Code Playground

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const filteredWords = words.filter(function (word) { return word.length > 6; });

console.log(filteredWords);
// expected output: Array ["exuberant", "destruction", "present"]
```

La función devuelve una nueva lista con los resultados, en lugar de alterar la lista original y es compatible con la convención establecida para React que promueve las estructuras de datos inmutables.

[75] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/filter

src/App.js

```
onDismiss(id) {
  const updatedList = this.state.list.filter(function isNotId(item) {
    return item.objectID !== id;
  });
}
```

En el siguiente paso, toma la función `isNotId()` que acabas de declarar y pasarla al método `filter()` como se muestra a continuación.

src/App.js

```
onDismiss(id) {
  function isNotId(item) {
    return item.objectID !== id;
  }

  const updatedList = this.state.list.filter(isNotId);
}
```

Recuerda: puedes filtrar más eficientemente utilizando una función flecha ES6.

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
}
```

Podrías incluso hacerlo todo en una sola línea, como hiciste con el selector `onClick()` del botón, aunque puede resultar difícil de leer:

src/App.js

```
onDismiss(id) {
  const updatedList = this.state.list.filter(item => item.objectID !== id);
}
```

La lista ahora remueve el elemento al que se le dio clic, pero el estado aún no se actualiza. Usa el método `setState()` de la clase para actualizar la lista en el estado local del componente.

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
  this.setState({ list: updatedList });
}
```

Ahora ejecuta nuevamente la aplicación y prueba el botón "Dismiss". Debe funcionar correctamente. Esto que acabas de experimentar se conoce como **Flujo de Datos Unidireccional**. Ejecutas una acción en la capa de vistas con `onClick()`, luego una función o método de clase modifica el estado interno del componente y acto seguido el respectivo método `render()` es ejecutado para actualizar la capa de vistas.

Ejercicios:

- lee más sobre el estado y los ciclos de vida en componentes React[76]

[76] https://facebook.github.io/react/docs/state-and-lifecycle.html

Enlaces (Bindings)

Es importante que conozcas los Enlaces que tienen lugar dentro de las clases JavaScript ES6 al momento de utilizar componentes de clase React ES6. En el capítulo anterior enlazaste el método de clase `onDismiss()` al constructor de la clase `App`.

src/App.js

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onDismiss = this.onDismiss.bind(this);
  }

  ...
}
```

Enlazar es necesario, porque los métodos de clase no enlazan `this` de manera automática a la instancia de la clase. Veamos esta idea en acción con la ayuda del siguiente componente de clase ES6.

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

El componente se renderiza sin problema, pero cuando presiones el botón, recibirás el mensaje `undefined` en la consola de desarrollador. Esta es una de las principales fuentes de bugs en React, pues, si quieres acceder a `this.state` desde un método de clase, esto no será posible porque `this` es `undefined` por defecto. Para poder acceder a `this` desde tus métodos de clase tienes que enlazar `this` a los métodos de clase.

En el siguiente componente de la clase, el método de la clase asociado apropiadamente en el constructor de la clase:

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.onClickMe = this.onClickMe.bind(this);
  }

  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

Enlazar métodos de clase puede hacerse desde cualquier otra parte también. Como por ejemplo, dentro del método de clase `render()`:

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe.bind(this)}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

Evita esta práctica, porque enlaza el método de la clase cada vez que se ejecuta `render()`, esto significa que actualiza el componente, lo que compromete el rendimiento. Al momento de enlazar un método de clase al constructor debes enlazarlo al principio, sólo una vez cuando el componente es instado.

Algunos desarrolladores definen la lógica de negocios de sus métodos de clase dentro del constructor:

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.onClickMe = () => {
      console.log(this);
    }
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
```

```
    );
  }
}
```

Debes evitarlo también. Con el tiempo esto desordenará el constructor. El constructor sólo está allí para que sea posible instar la clase y todas sus propiedades. Es por eso que la lógica de negocio de los métodos de clase debe ser definida fuera del constructor.

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.doSomething = this.doSomething.bind(this);
    this.doSomethingElse = this.doSomethingElse.bind(this);
  }

  doSomething() {
    // do something
  }

  doSomethingElse() {
    // do something else
  }

  ...
}
```

Los métodos de clase pueden auto-enlazarse utilizando funciones flecha de ES6:

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe = () => {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
```

```
        Click Me
      </button>
    );
  }
}
```

Si realizar enlaces dentro de los constructores repetidas veces te resulta molesto, puedes hacerlo de la manera antes mencionada. La documentación oficial de React sugiere que se enlacen los métodos de clase dentro del constructor, por eso el libro adoptará este enfoque también.

Ejercicios:

- Prueba los diferentes tipos de enlace mencionados anteriormente y registra en la consola de desarrollador (console.log) el objeto `this`
- Aprende más sobre una sintaxis alternativa del componente de React[77]

[77] https://github.com/the-road-to-learn-react/react-alternative-class-component-syntax

Controlador de Eventos (Event Handler)

En esta sección adquirirás un mayor entendimiento acerca de los controladores de eventos. Dentro de tu aplicación, estas utilizando el siguiente elemento `button` para eliminar un elemento de la lista.

src/App.js

```
...

<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

Esto ya representa un caso de uso complejo porque tienes que pasar un valor al método de clase y por lo tanto, debes colocarlo dentro de otra función flecha. Básicamente, lo que debe ser pasado al controlador de evento es una función. El siguiente código no funcionaría, el método de clase sería ejecutado inmediatamente al abrir la aplicación dentro del navegador.

src/App.js

```
...

<button
  onClick={this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

Al usar `onClick={doSomething()}`, la función `doSomething()` se ejecutaría de inmediato al momento de abrir la aplicación en el navegador. La expresión pasada al controlador es evaluada. Y como el valor retornado por la función no es una función, nada pasaría al presionar el botón. Pero al utilizar `onClick={doSomething}`, donde `doSomething` es una función, ésta sería ejecutada al momento de presionar el botón. Y la misma regla aplica para el método de clase `onDismiss()` que es usado en tu aplicación.

Sin embargo, utilizar `onclick={this.onDismiss}` no sería suficiente, porque de alguna manera la propiedad `item.objectID` debe ser pasada al método de clase para identificar el elemento que va

a ser eliminado. Por eso puede ser colocado como parámetro dentro de otra función y acceder a la propiedad. A este concepto se le conoce cómo función de orden superior en JavaScript y será explicado más adelante.

src/App.js

```
...

<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

Una alternativa sería definir la función envolvente fuera en otro lugar y sólo pasar la función ya definida al controlador. Como necesita acceso al elemento individual, tiene que habitar dentro del bloque de la función map.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item => {
          const onHandleDismiss = () =>
            this.onDismiss(item.objectID);

          return (
            <div key={item.objectID}>
              <span>
                <a href={item.url}>{item.title}</a>
              </span>
              <span>{item.author}</span>
              <span>{item.num_comments}</span>
              <span>{item.points}</span>
              <span>
                <button
                  onClick={onHandleDismiss}
```

```
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        );
      }
      )}
    </div>
  );
 }
}
```

Después de todo, debe ser una función lo que es pasado al controlador del elemento. Como ejemplo, prueba este código:

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
            ...
            <span>
              <button
                onClick={console.log(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Este método arrancará al abrir la aplicación en el navegador, pero no cuando presiones el botón. Mientras que la siguiente pieza de código sólo correrá cuando presiones el botón. Es decir, la función será ejecutada cuando acciones el controlador.

src/App.js

```
...

<button
  onClick={function () {
    console.log(item.objectID)
  }}
  type="button"
>
  Dismiss
</button>

...
```

Recuerda: Puedes transformar funciones a una función flecha JavaScript ES6. Similar a lo que hicimos con el método de clase `ondismiss()`.

src/App.js

```
...

<button
  onClick={() => console.log(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

Es común que sea complicado para algunos principiantes utilizar funciones dentro de un controlador de eventos, pero no te desanimes si tienes problemas en el primer intento. Al final deberás tener una función flecha de una sola línea, que además puede acceder a la propiedad `objectID` del objeto `item`:

src/App.js

```
class App extends Component {
  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
          <div key={item.objectID}>
            ...
            <span>
              <button
                onClick={() => this.onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Otro tema relevante en cuanto a rendimiento es la implicación de utilizar funciones flecha dentro de controladores de eventos es, por ejemplo, el controlador `onClick` envuelve al método `onDismiss()` dentro de una función flecha para así poder captar el identificador del elemento. Así cada vez que el método `render()` se ejecuta, el controlador insta una función flecha de orden superior. Esto puede impactar de cierta manera el rendimiento de tu aplicación, pero en la mayoría de los casos no se notará. Imagina que tienes una gran tabla de datos con 1000 elementos y cada fila o columna posee dicha función flecha dentro de su controlador de evento, en este caso vale la pena pensar en las implicaciones de rendimiento y por lo tanto podrías implementar un componente Botón dedicado a enlazar el método dentro del constructor. Pero antes de que eso suceda es una optimización prematura. Por ahora basta con que te enfoques meramente en aprender React.

Ejercicios:

- Experimenta con diferentes formas de utilizar funciones dentro del controlador `onClick` de tu botón

Interacciones con Formularios y Eventos

Añadiremos otra interacción relacionada con formularios y eventos en React, una funcionalidad de búsqueda donde la entrada del campo de búsqueda se debe filtra una lista basada en la propiedad de título de un elemento.

Primero, define el campo de entrada en tu JSX:

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input type="text" />
        </form>
        {this.state.list.map(item =>
          ...
        )}
      </div>
    );
  }
}
```

En el escenario siguiente, escribirás dentro del campo establecido y se filtrará la lista temporalmente de acuerdo al término de búsqueda especificado. Para filtrar la lista almacenamos el valor del campo de entrada en el estado local. En React puedes utilizar los **eventos sintéticos (synthetic events)** para acceder al valor que necesitas de este evento.

Vamos a definir un manejador `onChange()` para el campo de entrada.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        ...
      </div>
    );
  }
}
```

La función está vinculada al componente y, por tanto se le considera un método de clase. Sólo tienes que vincularlo y definirlo:

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  onSearchChange() {
    ...
  }
```

```
  ...
}
```

Al usar un controlador dentro de tu elemento, obtienes acceso al evento sintético de React dentro de la función callback.

src/App.js

```
class App extends Component {

  ...

  onSearchChange(event) {
    ...
  }

  ...
}
```

El evento tiene el valor del campo de entrada en su objeto de destino, por lo que es posible actualizar el estado local con el término de búsqueda utilizando `this.setState()`.

src/App.js

```
class App extends Component {

  ...

  onSearchChange(event) {
    this.setState({ searchTerm: event.target.value });
  }

  ...
}
```

No olvides definir el estado inicial para la propiedad `searchTerm` dentro del constructor de la clase. El campo de entrada estará vacío al principio y por lo tanto el valor de `searchTerm` debe ser una cadena de texto vacía (empty string).

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
      searchTerm: '',
    };

    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...
}
```

El valor de entrada es almacenado dentro del estado de componente interno cada vez que el valor en el campo de entrada cambia.

Podemos asumir que al actualizar `searchTerm` con `this.setState()`, la lista debe ser pasada también con el fin de preservarla. Sin embargo, `this.setState()` de React es una unión superficial, por lo que preserva las propiedades de su hermano dentro del estado del objeto al momento de actualizar una propiedad específica en él. El estado de la lista, aunque ya ha descartado un elemento de la misma, se mantiene igual al actualizar la propiedad `searchTerm`.

De vuelta a la aplicación, la lista aún no está filtrada con base en la información del campo de entrada que se almacena en el estado local del componente. Básicamente, se busca filtrar la lista de manera temporal en base al elemento `searchTerm` y tenemos lo necesario para lograrlo. Dentro del método `render()`, antes de mapear la lista, le aplicaremos un filtro. Este filtro sólo evaluaría si `searchTerm` coincide con el título de propiedad del elemento. Ya utilizaste anteriormente el método `filter` de JavaScript, aquí lo utilizarás nuevamente. Es posible agregar en primer lugar la función `filter` antes que `map`, porque `filter` retorna un nuevo array, por lo que la función `map` resulta ser muy conveniente en esta ocasión.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        {this.state.list.filter(...).map(item =>
          ...
        )}
      </div>
    );
  }
}
```

Ahora analicemos la función `filter` desde otro punto de vista. Queremos definir el argumento de `filter` (la función que es pasada como parámetro a `filter`) fuera de nuestro componente de clase ES6. Desde allí no se tiene acceso al estado del componente y por lo tanto no tenemos acceso a la propiedad `searchTerm` para evaluar la condición del filtro. Tenemos que pasar `searchTerm` a la función de filtro y esta tiene que devolver una nueva función para evaluar la condición. A esta nueva función se le conoce como función de orden superior (higher-order function).

Normalmente no mencionaría las funciones de orden superior, pero en un libro que habla acerca de React, esto es importante. Es necesario conocer sobre las funciones de orden superior porque React trata con un concepto llamado componentes de orden superior. Este concepto se explora más adelante en el libro. Por ahora, volvamos a enfocarnos en la función `filter` y su funcionalidad.

Primero, tienes que definir la función de orden superior fuera del componente `App`.

src/App.js

```
function isSearched(searchTerm) {
  return function (item) {
    // some condition which returns true or false
  }
}

class App extends Component {

  ...

}
```

La función `isSearched()` toma a `searchTerm` como parámetro y devuelve otra función, porque la función `filter` únicamente toma ese tipo como entrada. La función devuelta tiene acceso al elemento del objeto porque es la función que es pasada como parámetro a la función `filter`.

Adicionalmente, la función devuelta se utilizará para filtrar la lista en base a la condición definida dentro de la función. Vamos a definir la condición:

src/App.js

```
function isSearched(searchTerm) {
  return function (item) {
    return item.title.toLowerCase().includes(searchTerm.toLowerCase());
  }
}

class App extends Component {

  ...

}
```

La condición establece que el patrón entrante de `searchTerm` coincide con el título de la propiedad perteneciente al elemento de la lista. Esto se puede lograr con la funcionalidad `includes` de JavaScript. Sólo cuando el patrón coincide, se retorna verdadero y el ítem permanece dentro de la lista. Cuando el patrón no coincide el ítem es removido de la lista. Pero ten cuidado cuando los patrones coinciden. No debes olvidar convertir a minúsculas ambas cadenas de texto. De otro modo habrán inconsistencias entre un término de búsqueda 'redux' y un ítem con el título 'Redux'. Ya que estamos trabajando con una lista inmutable que retorna una nueva lista al usar la función `filter()`, la lista original almacenada en el estado local no es modificada en lo absoluto.

Resta algo por mencionar: Hicimos un poco de trampa utilizando características de JavaScript ES6 que no está en ES5. Para JavaScript ES5 podrías utilizar la función `indexOf()` para obtener el índice del elemento en la lista, cuando el elemento se encuentre en la lista `indexOf()` retornará su índice en el arreglo.

Code Playground

```
// ES5
string.indexOf(pattern) !== -1

// ES6
string.includes(pattern)
```

Otra refactorización inteligente puede ser lograda utilizando de nuevo una función flecha ES6. Hace que la función sea más concisa:

Code Playground

```
// ES5
function isSearched(searchTerm) {
  return function (item) {
    return item.title.toLowerCase().indexOf(searchTerm.toLowerCase()) !== -1;
  }
}

// ES6
const isSearched = searchTerm => item =>
  item.title.toLowerCase().includes(searchTerm.toLowerCase());
```

El ecosistema de React utiliza una gran cantidad de conceptos de programación funcional. Sucede a menudo que puedes utilizar una función que devuelve otra función (funciones de orden superior) para pasar información. En ES6 estas se pueden expresar de forma más concisa utilizando funciones flecha de ES6.

Por último, pero no menos importante, tienes que utilizar la función definida `isSearched()` para filtrar tu lista. `isSearched` recibe como parámetro la propiedad `searchTerm` directamente del estado local, retorna la entrada de la función `filter()` y filtra la lista de acuerdo a la condición del filtro. Después mapea la lista filtrada para mostrar en pantalla un elemento correspondiente a cada ítem.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        {this.state.list.filter(isSearched(this.state.searchTerm)).map(item =>
          ...
        )}
      </div>
    );
  }
}
```

Ahora la funcionalidad de búsqueda debería trabajar correctamente. Pruébala en tu navegador.

Ejercicios:

- lee más sobre eventos React[78]
- lee más sobre Funciones de Orden Superior[79]

[78] https://facebook.github.io/react/docs/handling-events.html
[79] https://es.wikipedia.org/wiki/Funci%C3%B3n_de_orden_superior

Desestructuración ES6 (Destructuring)

En ES6 es posible acceder a las propiedades de objetos y arreglos fácilmente, se le conoce como desestructuración. Compara los siguientes fragmentos de JavaScript ES5 y ES6.

Code Playground

```
const user = {
  firstname: 'Robin',
  lastname: 'Wieruch',
};

// ES5
var firstname = user.firstname;
var lastname = user.lastname;

console.log(firstname + ' ' + lastname);
// output: Robin Wieruch

// ES6
const { firstname, lastname } = user;

console.log(firstname + ' ' + lastname);
// output: Robin Wieruch
```

Mientras que en ES5 tienes que agregar una línea extra cada vez que desees acceder a la propiedad de objeto en ES6 puedes hacerlo en una sola línea. Una buena práctica al desestructurar un objeto es utilizar múltiples líneas para mejorar la legibilidad en un objeto con múltiples propiedades.

Code Playground

```
const {
  firstname,
  lastname
} = user;
```

Lo mismo aplica para los arreglos. También puedes desestructurarlos y mantenerlos legibles utilizando múltiples líneas para representar multiples propiedades.

Code Playground

```
const users = ['Robin', 'Andrew', 'Dan'];
const [
  userOne,
  userTwo,
  userThree
] = users;

console.log(userOne, userTwo, userThree);
// output: Robin Andrew Dan
```

Quizá notaste que el estado en el componente `App` puede desestructurarse de la misma manera. Se puede acortar el filtro y la línea map del código.

src/App.js

```
  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        ...
        {list.filter(isSearched(searchTerm)).map(item =>
          ...
        )}
      </div>
    );
```

Puedes hacerlo a la manera ES5 o ES6:

Code Playground

```
// ES5
var searchTerm = this.state.searchTerm;
var list = this.state.list;

// ES6
const { searchTerm, list } = this.state;
```

En el libro se utiliza JavaScript ES6 la mayor parte del tiempo, debes apegarte a ES6.

Ejercicios:

- lee más sobre desestructuración ES6[80]

[80] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/Destructuring_assignment

Componentes Controlados

Anteriormente aprendiste sobre el flujo de datos unidireccional en React. La misma ley se aplica al campo de entrada, que actualiza el estado local con el `searchTerm` para filtrar la lista. Al momento en que el estado cambia, el método `render()` se ejecuta nuevamente y utiliza nuevamente `searchTerm` almacenado en el estado local para aplicar la condición de filtrado.

¿Pero no nos olvidamos de algo en el campo de entrada? Una etiqueta de entrada (input) HTML incluye un atributo `value`. El atributo `value` normalmente almacena el valor suministrado en el campo de entrada. En este caso sería la propiedad `searchTerm`. Elementos de formulario como `<input>`, `<textarea>` y `<select>` mantienen su propio estado en HTML simple. Modifican el valor internamente una vez que alguien lo cambia desde el exterior. En React se les llama **componentes no controlados**, porque manejan su propio estado. En React, debe asegurarse de que estos elementos sean **componentes controlados**.

Para lograr esto, establecemos el atributo `value` del campo de entrada que se encuentra almacenado en la propiedad de estado `searchTerm` para que podamos acceder a éste desde allí:

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <form>
          <input
            type="text"
            value={searchTerm}
            onChange={this.onSearchChange}
          />
        </form>
        ...
      </div>
    );
  }
}
```

El ciclo de flujo de datos unidireccional para el campo de entrada es autónomo ahora. El estado interno del componente es la única fuente de información para el campo de entrada.

Toda la gestión del estado local y el flujo de datos unidireccional podría ser nuevo para ti. Pero una vez que te acostumbres a él, será la forma natural de implementar elementos en React. En general, React ofrece un nuevo patrón con el flujo de datos unidireccional al mundo de las aplicaciones de una sola página. Este patrón es adoptado por varios frameworks y librerías.

Ejercicios:

- lee más sobre React forms[81]
- Aprende más sobre diferentes componentes controlados[82]

[81] https://facebook.github.io/react/docs/forms.html
[82] https://github.com/the-road-to-learn-react/react-controlled-components-examples

Dividir Componentes

Ahora tienes un componente de App grande que sigue creciendo y eventualmente será demasiado complejo para administrar efectivamente. Necesitamos dividirlo en partes más pequeñas, más manejables mediante la creación de componentes separados para la entrada de búsqueda y la lista de elementos.

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search />
        <Table />
      </div>
    );
  }
}
```

Puedes pasar las propiedades de los componentes que pueden utilizar ellos mismos. El componente `App` necesita pasar las propiedades administradas en el estado local y sus métodos de clase.

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search
          value={searchTerm}
          onChange={this.onSearchChange}
        />
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
```

```
      </div>
    );
  }
}
```

Ahora puedes definir los componentes junto a su componente App. Estos componentes serán también componentes de la clase ES6. Renderizan los mismos elementos como antes.

El primero es el componente de búsqueda.

```
class App extends Component {
  ...
}

class Search extends Component {
  render() {
    const { value, onChange } = this.props;
    return (
      <form>
        <input
          type="text"
          value={value}
          onChange={onChange}
        />
      </form>
    );
  }
}
```

El segundo es el componente Tabla.

```
...

class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        { list.filter(isSearched(pattern)).map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
```

```
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <button
                onClick={() => onDismiss(item.objectID)}
                type="button"
                >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Ahora tienes tres componentes de clase ES6. Nota que el objeto `props` es accesible a través de la instancia de la clase usando `this`. Las props - abreviatura para las propiedades - tienen todos los valores que han pasado a los componentes cuando los utiliza en su componente App. Podrías reutilizar estos componentes en otro lugar, pero pasarles diferentes valores. Son reutilizables.

Ejercicios:

- averigua qué componentes podrías dividir como los elementos Search y Table, pero espera a que cubramos más de estos conceptos antes de implementarlos.

Componentes Ensamblables

La propiedad `hijoes` usada para pasar elementos a sus componentes desde arriba, que son desconocidos para el componente mismo, pero hacen posible ensamblar componentes entre sí. Veamos cómo se ve esto cuando sólo pasa un texto (string) como hijo al componente Search.

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search
          value={searchTerm}
          onChange={this.onSearchChange}
        >
          Search
        </Search>
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

Ahora, el componente de búsqueda puede desestructurar la propiedad `children` del objeto `props` y especificar dónde deben mostrarse los hijos.

```
class Search extends Component {
  render() {
    const { value, onChange, children } = this.props;
    return (
      <form>
        {children} <input
          type="text"
          value={value}
          onChange={onChange}
        />
      </form>
    );
  }
}
```

Ahora el texto “Search” debe estar visible al lado de su campo de entrada. Cuando utilices el componente Search en algún otro lugar, puedes elegir un texto diferente si tú quieres. Después de todo, no es sólo el texto que puede pasar como hijos. Puedes pasar un elemento y arboles de elementos (que puede ser encapsulado por los componentes de nuevo) como hijos. La propiedad hijo hace posible tejer componentes entre sí.

Ejercicios:

- leer más sobre el modelo de composición de React[83]

[83] https://facebook.github.io/react/docs/composition-vs-inheritance.html

Componentes Reutilizables

Los componentes compuestos(composable) te dan poder para crear jerarquías de componentes capaces. Son la base de su capa de vista. Los últimos capítulos mencionan a menudo el término reutilización. Puedes volver a utilizar los componentes Table y Search. Incluso el componente `App` es reutilizable, pues puede ser instado desde cualquier otra parte también.

Vamos a definir un componente más reutilizable - un componente Button - que eventualmente se reutiliza más a menudo.

```
class Button extends Component {
  render() {
    const {
      onClick,
      className,
      children,
    } = this.props;

    return (
      <button
        onClick={onClick}
        className={className}
        type="button"
      >
        {children}
      </button>
    );
  }
}
```

Podría parecer redundante declarar un componente como éste. Usarás un componente `Button` en lugar de un elemento `button`. Sólo ahorra el `type="button"`. A excepción del atributo type, debes definir todo lo demás cuando desees utilizar el componente Button. Pero hay que pensar en la inversión a largo plazo aquí. Imagina que tienes varios botones en tu aplicación, pero deseas cambiar un atributo, estilo o comportamiento para el botón. Sin el componente tendrías que refactorizar cada botón. En cambio, el componente Button asegura tener una fuente única de verdad. Un Button para refactorizar todos los botones a la vez. Un Button para gobernarlos a todos.

Ya que ya tienes un elemento button, puedes utilizar el componente Button en su lugar. Omite el atributo type.

```
class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        { list.filter(isSearched(pattern)).map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <Button onClick={() => onDismiss(item.objectID)}>
                Dismiss
              </Button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

El componente Button espera una propiedad `className` en las `props`. El atributo `className` es otro derivado de React para el atributo class de HTML. Pero no pasamos ningún atributo `className` cuando Button fue usado. En el código debe ser más explícito en el componente Button que el `className` es opcional, por lo tanto puedes utilizar un valor por defecto en tu desestructuración del objeto.

```
class Button extends Component {
  render() {
    const {
      onClick,
      className = '',
      children,
    } = this.props;

    ...
  }
}
```

Ahora, cuando no haya ninguna propiedad `className` cuando se use el componente Button, el valor será una cadena vacía en lugar de `undefined`.

Ejercicios:

- leer más sobre parámetros por defecto de ES6[84]

[84] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Funciones/Parametros_por_defecto

Declaraciones de componentes

Por ahora tienes cuatro componentes de clase ES6. Pero puedes hacerlo mejor. Permíteme introducir componentes funcionales sin estado como alternativa para componentes de clase ES6. Antes de refactorizar sus componentes, vamos a introducir los diferentes tipos de componentes.

- **Componentes funcionales sin estado** Estos componentes son funciones que reciben una entrada y devuelven una salida. La entrada es el objeto props. La salida es una instancia de componente, por lo tanto JSX simple. Hasta ahora es bastante similar a un componente de clase ES6. Sin embargo, los componentes funcionales sin estados son funciones (funcional) y no tiene un estado interno (stateless). No puedes acceder o actualizar el estado con `this.state` o `this.setState()` porque no hay un objeto `this`. Además, no tienen métodos de ciclo de vida excepto por el método `render` que será aplicado implícitamente en los componentes sin estados funcionales. Todavía no has aprendido los métodos del ciclo de vida, pero ya usaste dos: `constructor()` y `render()`. El constructor se ejecuta sólo una vez en la vida de un componente, mientras el método `render()` de la clase se ejecuta una vez al principio y siempre que el componente se actualiza. Mantén este hecho en mente sobre los componentes funcionales sin estado, cuando llegues al capítulo de métodos de ciclo de vida más adelante.
- **Componentes de clase ES6** extiende del componente React. El `extend` engancha todos los métodos del ciclo de vida, disponibles en la API del componente React, al componente. Así es como pudimos utilizar el método de la clase `render()`. También puedes almacenar y manipular el estado en componentes clases de ES6 usando `this.state` y `this.setState()`.
- **React.createClass** Esta declaración de componente se utilizó en versiones anteriores de React y aún se utiliza en aplicaciones de JavaScript ES5 React. Pero Facebook lo declaró obsoleto[85] en favor de ES6. Incluso añadieron un Advertencia de depreciación en la versión 15.5[86]. No lo usarás en el libro.

Al momento de decidir cuándo utilizar componentes funcionales sin estados en lugar de componentes clase de ES6, una regla general es usar componentes funcionales sin estado cuando no se necesitan métodos de ciclo de vida de componentes o estados locales. Por lo general, comienzas a implementar tus componentes como componentes funcionales sin estado. Una vez que necesite acceder a los métodos de estado o de ciclo de vida, debe refactorizarlo a un componente de clase ES6. Anteriormente iniciamos al en sentido opuesto en tu aplicación con el propósito de aprender.

Volvamos a tu aplicación. El componente App utiliza el estado local. Es por eso que tiene que permanecer como un componente de clase ES6. Pero los otros tres componentes de su clase ES6 son sin estado, pues no necesitan acceder a `this.state` o `this.setState()` y no tienen métodos de ciclo de vida. Vamos a refactorizar juntos el componente de búsqueda a un componente funcional sin estado. La refactorización del componente Tabla y botón serán tu ejercicio.

[85] https://facebook.github.io/react/blog/2015/03/10/react-v0.13.html
[86] https://facebook.github.io/react/blog/2017/04/07/react-v15.5.0.html

```
function Search(props) {
  const { value, onChange, children } = props;
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

Las `props` son accesibles en función firma y el valor que regresa es JSX. Pero puedes hacer más código inteligente en un componente funcional sin estado. Ya conoces la desestructuración ES6. La mejor práctica es utilizarla en la firma de funciones para desestructurar las `props`:

```
function Search({ value, onChange, children }) {
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

Recuerda que las funciones de flecha ES6 te permiten mantener tus funciones concisas. Puedes quitar el cuerpo del bloque de la función. En un cuerpo conciso un retorno implícito se adjunta así que puedes quitar la declaración `return`. Dado que su componente funcional sin estado es una función, puedes mantenerla concisa también.

```
const Search = ({ value, onChange, children }) =>
  <form>
    {children} <input
      type="text"
      value={value}
      onChange={onChange}
    />
  </form>
```

El último paso es especialmente útil para forzar props como entrada y un elemento JSX como salida. Sin embargo, podrías *hacer algo* usando un cuerpo del bloque en su función de la flecha ES6:

```
const Search = ({ value, onChange, children }) => {

  // do something

  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

Ahora tienes un componente funcional ligero sin estado. Una vez que necesites tener acceso a su estado de componente interno o métodos de ciclo de vida, lo refactorizarías a un componente de clase ES6. Al usar cuerpos de bloque, los programadores tienden a hacer sus funciones más complejas, pero dejar el cuerpo del bloque te permite enfocarte en la entrada y la salida. JavaScript ES6 en los componentes React los hace más elegantes y fáciles de leer.

Ejercicios:

- Refactorizar el componente de tabla y botón a componentes funcionales sin estado
- leer más sobre ES6 class components and functional stateless components[87]

[87] https://facebook.github.io/react/docs/components-and-props.html

Estilización de Componentes

En esta sección agregaremos un estilo básico a nuestra aplicación y sus componentes. Puedes reutilizar los archivos *src/App.css* y *src/index.css*. Estos archivos deben estar ya en su proyecto desde que lo iniciaste con *create-react-app*. Tambien deben ser importados en sus archivos *src/App.js* y *src/index.js*. He preparado algunos CSS que puede simplemente copiar y pegar en estos archivos, pero no dudes en utilizar tu propio estilo.

```
body {
  color: #222;
  background: #f4f4f4;
  font: 400 14px CoreSans, Arial,sans-serif;
}

a {
  color: #222;
}

a:hover {
  text-decoration: underline;
}

ul, li {
  list-style: none;
  padding: 0;
  margin: 0;
}

input {
  padding: 10px;
  border-radius: 5px;
  outline: none;
  margin-right: 10px;
  border: 1px solid #dddddd;
}

button {
  padding: 10px;
  border-radius: 5px;
  border: 1px solid #dddddd;
  background: transparent;
  color: #808080;
```

```
  cursor: pointer;
}

button:hover {
  color: #222;
}

*:focus {
  outline: none;
}


.page {
  margin: 20px;
}

.interactions {
  text-align: center;
}

.table {
  margin: 20px 0;
}

.table-header {
  display: flex;
  line-height: 24px;
  font-size: 16px;
  padding: 0 10px;
  justify-content: space-between;
}

.table-empty {
  margin: 200px;
  text-align: center;
  font-size: 16px;
}

.table-row {
  display: flex;
  line-height: 24px;
  white-space: nowrap;
  margin: 10px 0;
```

```
  padding: 10px;
  background: #ffffff;
  border: 1px solid #e3e3e3;
}

.table-header > span {
  overflow: hidden;
  text-overflow: ellipsis;
  padding: 0 5px;
}

.table-row > span {
  overflow: hidden;
  text-overflow: ellipsis;
  padding: 0 5px;
}

.button-inline {
  border-width: 0;
  background: transparent;
  color: inherit;
  text-align: inherit;
  -webkit-font-smoothing: inherit;
  padding: 0;
  font-size: inherit;
  cursor: pointer;
}

.button-active {
  border-radius: 0;
  border-bottom: 1px solid #38BB6C;
}
```

Ahora puedes utilizar el estilo en algunos de sus componentes. No te olvides de utilizar `className` de React en lugar de `class` como atributo HTML.

Primero, aplícalo en tu componente de la clase App ES6:

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
          >
            Search
          </Search>
        </div>
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

Segundo, aplícalo en tu componente funcional sin estado Table:

```
const Table = ({ list, pattern, onDismiss }) =>
  <div className="table">
    { list.filter(isSearched(pattern)).map(item =>
      <div key={item.objectID} className="table-row">
        <span>
          <a href={item.url}>{item.title}</a>
        </span>
        <span>{item.author}</span>
        <span>{item.num_comments}</span>
        <span>{item.points}</span>
        <span>
          <Button
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
```

```
            Dismiss
          </Button>
        </span>
      </div>
    )}
  </div>
```

Ahora has estilizado tu aplicación y componentes con CSS básico. Como sabes, JSX mezcla HTML y JavaScript. Nadie podría discutir para agregar CSS en la mezcla también. Eso se llama estilo en línea. Puede definir objetos JavaScript y pasarlos al atributo de estilo de un elemento.

Mantengamos flexibles la anchura de columna de la tabla mediante el uso del estilo en línea.

```
const Table = ({ list, pattern, onDismiss }) =>
  <div className="table">
    { list.filter(isSearched(pattern)).map(item =>
      <div key={item.objectID} className="table-row">
        <span style={{ width: '40%' }}>
          <a href={item.url}>{item.title}</a>
        </span>
        <span style={{ width: '30%' }}>
          {item.author}
        </span>
        <span style={{ width: '10%' }}>
          {item.num_comments}
        </span>
        <span style={{ width: '10%' }}>
          {item.points}
        </span>
        <span style={{ width: '10%' }}>
          <Button
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
            Dismiss
          </Button>
        </span>
      </div>
    )}
  </div>
```

Está realmente alineado ahora. Define los objetos de estilo fuera de tus elementos para hacerlo más limpio:

```
const largeColumn = {
  width: '40%',
};

const midColumn = {
  width: '30%',
};

const smallColumn = {
  width: '10%',
};
```

Después de eso podrías utilizarlo en tus columnas: `<span style={smallColumn}>`.

En general, te encontrarás con diferentes opiniones y soluciones para el estilo en React. Justo utilizaste puro estilo CSS y en línea ahora. Es suficiente para empezar.

No quiero ser dogmático aquí, pero quiero dejarte más opciones. Puedes leer sobre ellos y aplicarlos por su cuenta. Pero si eres nuevo en React, te recomendaría que te mantengas puro estilo CSS y en línea por ahora.

- styled-components[88]
- CSS Modules[89] (lee mi breve artículo sobre cómo usar módulos CSS en create-react-app[90])
- Sass[91] (lee mi breve artículo sobre cómo usar Sass en create-react-app[92])

Pero si eres nuevo en React, te recomiendo que te adhieras a CSS puro y el estilo en línea por ahora.

¡Has aprendido los fundamentos de React que te permitirán escribir tus propias aplicaciones! Repasemos los últimos capítulos:

- React
  - Usar `this.state` y `setState()` para administrar el estado del componente interno
  - Pasar funciones o métodos clases a tu controlador de elementos
  - Utilizar formularios y eventos en React para agregar interacciones
  - Flujo de datos unidireccional es un concepto importante en React
  - Adoptar componentes controlados
  - Declarar componentes con hijos y componentes reutilizables
  - Uso e implementación de componentes de clase ES6 y componentes funcionales sin estado
  - Enfoques para diseñar sus componentes

[88] https://github.com/styled-components/styled-components
[89] https://github.com/css-modules/css-modules
[90] https://www.robinwieruch.de/create-react-app-css-modules/
[91] https://github.com/css-modules/css-modules
[92] https://www.robinwieruch.de/create-react-app-with-sass-support/

- ES6
  - Funciones que están enlazadas a una clase son métodos de clase
  - Desestructuración de objetos y arreglos
  - Parámetros predeterminados
- General
  - Funciones de orden superior

Una vez más, tiene sentido tomar un descanso. Interiorizar lo aprendido y aplicarlo por ti mismo. Puedes experimentar con el código fuente que has escrito hasta ahora. También puedes conocer más en la documentación oficial[93].

El código fuente está disponible en el repositorio oficial[94].

[93] https://facebook.github.io/react/docs/installation.html
[94] https://github.com/rwieruch/hackernews-client/tree/2705dcd1a2027c4a6ecb8132428b399785afdfa5

Trabajar con una API real

Ahora es el momento de trabajar en serio con una API. Si no estás familiarizado, te recomiendo leer mi artículo sobre cómo llegué a conocer el mundo de las APIs[95].

Para nuestra primera aproximación al concepto usaremos Hacker News[96], un acumulador de noticias sobre tecnología. En este ejercicio utilizaremos la API de Hacker News para obtener las últimas noticias. Hay APIs básicas[97] y de búsqueda[98] para conseguir datos de la plataforma. En concreto usaremos la de búsqueda para encontrar las noticias.

Puedes visitar la información de la API en cualquier momento para obtener un mejor conocimiento de la estructura de los datos.

[95] https://www.robinwieruch.de/what-is-an-api-javascript/
[96] https://news.ycombinator.com/
[97] https://github.com/HackerNews/API
[98] https://hn.algolia.com/api

Métodos del ciclo de vida

Quizás recuerdes una breve mención de los métodos del ciclo de vida en el capítulo anterior, relacionado con el ciclo de vida de un componente de React. Pueden utilizarse en componentes de clase ES6, pero no en componentes funcionales sin estado (Stateless Functional Components)

Además del método `render()`, hay muchos otros métodos que pueden ser sobrescritos en un componente de clase React ES6. Todos estos son métodos del ciclo de vida.

Ya hemos cubierto dos métodos de ciclo de vida que se pueden usar en un componente de clase ES6: * El constructor sólo es llamado cuando se crea una instancia del componente y se inserta en el DOM. El componente es instanciado en un proceso llamado montaje. * El método `render()` también es llamado durante el proceso de montaje, además de cuando el componente actualiza. Cada vez que el estado o las props de un componente cambian, se llama al método `render()`.

Ahora ya sabes más acerca de estos dos métodos del ciclo de vida y cuándo se llaman. Ya los has utilizado. Pero hay más de ellos.

Hay dos métodos de ciclo de vida más cuando se monta un componente: `getDerivedStateFromProps()` y `componentDidMount()`. El constructor se llama primero, `getDerivedStateFromProps()` se llama antes del método `render()` y `componentDidMount()` se llama después del método `render()`.

En general, el proceso de montaje tiene 4 métodos de ciclo de vida. Se invocan en el siguiente orden:

- constructor()
- getDerivedStateFromProps()
- render()
- componentDidMount()

Pero, ¿qué pasa con la actualización del ciclo de vida de un componente que ocurre cuando el estado o las propiedades cambian? En general, tiene 5 métodos de ciclo de vida en el siguiente orden:

- componentWillReceiveProps()
- shouldComponentUpdate()
- componentWillUpdate()
- render()
- componentDidUpdate()

Por último, pero no menos importante, está el desmontaje del ciclo de vida. Sólo tiene un método de ciclo de vida: `componentWillUnmount()`.

No es necesario conocer todos estos métodos de ciclo de vida desde el principio. Puede ser intimidante, pero no usarás todos - incluso en una aplicación React madura. Sin embargo, es bueno saber que cada método del ciclo de vida se puede usar en casos particulares:

- **constructor(props)** - Se llama cuando el componente se inicializa. Puedes establecer un estado inicial del componente y vincular métodos de clase útiles durante ese método de ciclo de vida.
- **componentWillMount()** - Se llama antes del método del `render()`. Es por eso que podría ser utilizado para establecer el estado del componente interno, porque no activará una segunda renderización del componente. Generalmente se recomienda utilizar el `constructor()` para establecer el estado inicial.
- **render()** - Este método del ciclo de vida es **obligatorio** y devuelve los elementos como una salida del componente. El método debe ser puro y por lo tanto no debe modificar el estado del componente. Recibe como entrada propiedades (props) y estados (state) y regresa un elemento.
- **componentDidMount()** - Se llama una sola vez cuando el componente fue montado. Ese es el método perfecto para realizar una solicitud asíncrona para obtener datos de una API. Los datos obtenidos se almacenan en el estado interno del componente para mostrarlos en el método `render()`.
- **componentWillReceiveProps(nextProps)** - Se llama durante la actualización de un ciclo de vida. Como entrada recibirá las siguientes props . Puedes comparar las props siguientes con las anteriores (`this.props`) para aplicar un comportamiento diferente basado en la diferencia. Además, puede establecer el estado en función de las siguientes props.
- **shouldComponentUpdate(nextProps, nextState)** - Siempre se llama cuando el componente se actualiza debido a cambios de estado o props. Lo usarás en aplicaciones de React maduras para optimizaciones de rendimiento. Dependiendo de un booleano que regrese de este método de ciclo de vida, el componente y todos sus hijos se renderizarán o no en la actualización de un ciclo de vida. Puedes evitar que se ejecute el método de ciclo de vida render de un componente.
- **componentWillUpdate(nextProps, nextState)** - El método del ciclo de vida se invoca antes del método `render()`. Ya se disponen las siguientes props y el próximo estado. Se puede utilizar el método como última oportunidad para realizar las preparaciones antes de ejecutar el método render. Hay que tener en cuenta que ya no se puede activar `setState()`. Si deseas calcular el estado basado en las siguientes props, tienes que usar `componentWillReceiveProps()`.
- **componentDidUpdate(prevProps, prevState)** - El método del ciclo de vida se invoca inmediatamente después del método `render()`. Puedes usarlo como oportunidad para realizar operaciones DOM o para realizar más solicitudes asíncronas.
- **componentWillUnmount()** - Se llama antes de destruir tu componente. Puedes utilizar el método del ciclo de vida para realizar tareas de limpieza.

Ya has utilizado los métodos `constructor()` y `render()`. Estos son los comúnmente utilizados por componentes de clase ES6. Hay que recordar que el método `render()` es necesario, de lo contrario no devolvería una instancia del componente.

Ejercicios:

- Leer mas sobre Ciclos de vida en React (En ingles)[99]
- Leer mas sobre Estado y Ciclo de vida en React (En ingles)[100]

[99] https://facebook.github.io/react/docs/react-component.html
[100] https://facebook.github.io/react/docs/state-and-lifecycle.html

Obteniendo Datos

Ahora estás listo para obtener datos de la API de Hacker News. He mencionado un método de ciclo de vida que se puede utilizar para obtener datos: `componentDidMount()`, donde se utilizará la API nativa para realizar la solicitud.

Antes de poder usarlo, vamos a configurar las constantes de url y los parámetros por defecto para dividir la solicitud de API en partes.

```
import React, { Component } from 'react';
import './App.css';

const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';

...
```

En JavaScript ES6 puedes usar plantillas de cadena de texto[101] para concatenar cadenas. Usaremos estas plantillas de cadenas de texto para armar la URL con la que vamos a conectarnos en la API (Endpoint)

```
// ES6
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${DEFAULT_QUERY}`;

// ES5
var url = PATH_BASE + PATH_SEARCH + '?' + PARAM_SEARCH + DEFAULT_QUERY;

console.log(url);
// output: https://hn.algolia.com/api/v1/search?query=redux
```

Esto ayudara a que la URL sea flexible en el tiempo.

Pero vayamos a la solicitud de API donde usarás la url. El proceso completo de búsqueda de datos se presentará de una vez, Pero cada paso se explicará después.

[101] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/template_strings

```
...

class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result: null,
      searchTerm: DEFAULT_QUERY,
    };

    this.setSearchTopstories = this.setSearchTopstories.bind(this);
    this.fetchSearchTopstories = this.fetchSearchTopstories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  setSearchTopstories(result) {
    this.setState({ result });
  }

  fetchSearchTopstories(searchTerm) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}`)
      .then(response => response.json())
      .then(result => this.setSearchTopstories(result))
      .catch(e => e);
  }

  componentDidMount() {
    const { searchTerm } = this.state;
    this.fetchSearchTopstories(searchTerm);
  }

  ...
}
```

Muchas cosas suceden en el código anterior. Pensé en mostrarlo en varias partes pero seria mas difícil comprender las relaciones entre cada pieza. Permíteme poder explicarte cada uno en detalle.

En primer lugar, puedes eliminar la lista artificial de elementos, porque obtenemos un resultado de la API de Hacker News. El estado inicial de su componente tiene un resultado vacío y un término de búsqueda predeterminado. El mismo término de búsqueda predeterminado se utiliza en el campo de búsqueda y en su primera solicitud.

En segundo lugar, utilizas el método `componentDidMount()` para obtener los datos después de que el componente se ha montado. En la primera búsqueda, se procederá a utilizar el termino de búsqueda predeterminado en el estado del componente. Por eso mismo, obtendrá historias relacionadas a "redux".

En tercer lugar, la búsqueda nativa se utiliza. Las cadenas de plantilla de JavaScript ES6 le permiten componer la url con el `searchTerm`. TLa url es el argumento de la función API de búsqueda nativa. La respuesta necesita ser transformada en json, es un paso obligatorio en una búsqueda nativa, y finalmente se puede establecer en el estado del componente interno.

Por último, pero no menos importante, no olvide vincular sus nuevos métodos de componentes.

Ahora puede utilizar los datos obtenidos en lugar de la lista artificial de elementos.Sin embargo, tienes que tener cuidado otra vez. El resultado no es sólo una lista de datos. Es un objeto complejo con meta información y una lista de éxitos (historias).[102] Puede emitir el estado interno con `console.log(this.state);` ent tu método `render()` para visualizarlo.

Utilicemos el resultado para mostrarlo. Pero lo preveremos de renderizar cualquier cosa - return null - cuando no hay resultado. Una vez que la solicitud a la API tuvo éxito, el resultado se guarda en el estado y el componente App se volverá a renderizar con el estado actualizado.

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;

    if (!result) { return null; }

    return (
      <div className="page">
        ...
        <Table
          list={result.hits}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

Repasemos lo que sucede durante el ciclo de vida del componente. El componente es inicializa por el constructor. Después de que se renderiza por primera vez. Pero evitas que muestre, porque el

[102] https://hn.algolia.com/api

resultado está vacío. Entonces se ejecuta el método `componentDidMount()`. En ese método, obtienes los datos de la API de Hacker News de forma asíncrona. Una vez que los datos llegan, cambia el estado interno del componente. Después de eso, el ciclo de vida de la actualización entra en juego. El componente ejecuta de nuevo el método `render()`, pero esta vez con datos poblados en su estado interno del componente. El componente y, por tanto, el componente Tabla con su contenido se vuelve a renderizar.

Utilizó la API de recuperación nativa que es compatible con la mayoría de los navegadores para realizar una solicitud asíncrona a una API. La configuración de *create-react-app* garantiza su compatibilidad con todos los navegadores. Hay paquetes de terceros de node que puedes usar para sustituir la API de búsqueda nativa: superagent[103] y axios[104].

Regresa a tu aplicación: La lista de hits debe ser visible ahora. Pero el botón "Dismiss" no funciona. Lo arreglaremos en el siguiente capitulo.

Ejercicios:

- leer mas sobre ES6 plantilas de cadena de texto[105]
- leer mas sobre the native fetch API[106]
- experimenta con Hacker News API[107]

[103] https://github.com/visionmedia/superagent
[104] https://github.com/mzabriskie/axios
[105] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/template_strings
[106] https://developer.mozilla.org/en/docs/Web/API/Fetch_API
[107] https://hn.algolia.com/api

ES6 Operadores de propagación

El botón "Dismiss" no funciona porque el método `onDismiss()` no tiene conocimiento del objeto complejo resultante. Cambiemos eso:

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    ...
  });
}
```

Pero, ¿qué pasa en `setState()` ahora? Desafortunadamente el resultado es un objeto complejo. La lista de hits es sólo una de las múltiples propiedades del objeto. Sin embargo, sólo la lista se actualiza, cuando un elemento se quita en el objeto resultante, mientras que las otras propiedades permanecen igual.

Un enfoque podría ser mutar los hits en el objeto resultante. Lo demostraré, pero no lo haremos de esa manera.

```
this.state.result.hits = updatedHits;
```

React abarca la programación funcional. Por lo tanto, no debes mutar un objeto (o mutar el estado directamente). Un mejor enfoque es generar un nuevo objeto basado en la información que tienes. De este modo, ninguno de los objetos se altera. Mantendrás las estructuras de datos inmutables. Siempre devolverás un objeto nuevo y nunca alterarás un objeto.

Vamos a hacerlo en JavaScript ES5. `Object.assign()` toma como primer argumento un objeto destino. Todos los argumentos siguientes son objetos de origen. Estos objetos se combinan en el objeto de destino. El objeto de destino puede ser un objeto vacío. Abarca inmutabilidad, porque ningún objeto fuente se muta. Sería similar a lo siguiente:

```
const updatedHits = { hits: updatedHits };
const updatedResult = Object.assign({}, this.state.result, updatedHits);
```

Ahora vamos a hacerlo en el método `onDismiss()`:

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    result: Object.assign({}, this.state.result, { hits: updatedHits })
  });
}
```

Eso es en JavaScript ES5. Existe una solución más sencilla en ES6 y futuras versiones de JavaScript. ¿Puedo presentarles el operador de propagación? Sólo consta de tres puntos: . . . Cuando se utiliza, cada valor de una matriz u objeto se copia a otra matriz u objeto.

Examinemos el operador de propagación de arreglos de ES6, aunque aún no lo necesites.

```
const userList = ['Robin', 'Andrew', 'Dan'];
const additionalUser = 'Jordan';
const allUsers = [ ...userList, additionalUser ];

console.log(allUsers);
// output: ['Robin', 'Andrew', 'Dan', 'Jordan']
```

La variables `allUsers` es un arreglo completamente nuevo. Las otras variables `userList` y `additionalUser` siguen igual. Incluso puedes combinar dos arreglos de esa manera en un nuevo arreglo.

```
const oldUsers = ['Robin', 'Andrew'];
const newUsers = ['Dan', 'Jordan'];
const allUsers = [ ...oldUsers, ...newUsers ];

console.log(allUsers);
// output: ['Robin', 'Andrew', 'Dan', 'Jordan']
```

Ahora echemos un vistazo al operador de propagación de objetos. ¡No es ES6! Es una propuesta para una futura versión de ES[108] ya utilizada por la comunidad React. Es por eso que *create-react-app* incorporó la característica en la configuración.

Básicamente es el mismo que el operador de ES6 de dispersión de array de JavaScript pero con objetos. Copia cada par de valores clave en un nuevo objeto.

[108] https://github.com/sebmarkbage/ecmascript-rest-spread

```
const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const age = 28;
const user = { ...userNames, age };

console.log(user);
// output: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }
```

Múltiples objetos pueden dispersarse como en el ejemplo de dispersión de arreglos.

```
const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const userAge = { age: 28 };
const user = { ...userNames, ...userAge };

console.log(user);
// output: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }
```

Después de todo se puede utilizar para reemplazar ES5 `Object.assign()`.

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    result: { ...this.state.result, hits: updatedHits }
  });
}
```

El botón "Dismiss" debería funcionar de nuevo.

Ejercicios:

- leer más sobre Object.assign()[109]
- leer más sobre el Operador de propagación de arreglos ES6[110]
  - el operador de propagación de objetos es mencionado brevemente

[109] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign
[110] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator

Renderizado Condicional

El renderizado condicional se introduce muy temprano en las aplicaciones React. Sucede cuando se quiere tomar la decisión de renderizar uno u otro elemento. A veces significa renderizar un elemento o nada. Después de todo, una representación condicional de uso más simple puede ser expresada por una sentencia if-else en JSX.

El objeto `resultante` en el estado de interno del componente es nulo al principio. Hasta el momento, el componente App no devuelve elementos cuando el resultado no ha llegado de la API. Eso ya es un renderizado condicional, porque vuelves antes del método `render()` por cierta condición. El componente App no renderiza nada o sus elementos.

Pero vamos un paso más allá. Tiene más sentido envolver el componente Table, que es el único componente que depende del resultado, en un renderizado condicional independiente. Todo lo demás se debe mostrar, aunque no hay ningún `resultado` aún. Simplemente puede utilizar una expresión ternaria en su JSX.

```
class App extends Component {

  ...

  render() {t
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
          >
            Search
          </Search>
        </div>
        { result
          ? <Table
            list={result.hits}
            pattern={searchTerm}
            onDismiss={this.onDismiss}
          />
          : null
        }
      </div>
    );
```

```
    }
  }
```

Esa es tu segunda opción para expresar un renderizado condicional. Una tercera opción es el operador `&&` lógico . En JavaScript, un `true && 'Hello World'` siempre se evalúa como 'Hello World'. Un `false && 'Hello World'` siempre se evalúa como falso.

```
const result = true && 'Hello World';
console.log(result);
// output: Hello World

const result = false && 'Hello World';
console.log(result);
// output: false
```

En React puedes hacer uso de ese comportamiento. Si la condición es verdadera, la expresión después del operador lógico `&&` será la salida. Si la condición es falsa, React ignora y omite la expresión. Es aplicable en el caso de renderizado condicional de la tabla, ya que debe devolver una Tabla o nada.

```
{ result &&
  <Table
    list={result.hits}
    pattern={searchTerm}
    onDismiss={this.onDismiss}
  />
}
```

Estos fueron algunos enfoques para utilizar el renderizado condicional en React. Puedes leer sobre más alternativas en mi sitio web[111] donde mantengo una lista exhaustiva de renderizaciones condicionales. Además, conocerá sus diferentes casos de uso y cuándo aplicarlos.

Después de todo, debería poder ver los datos obtenidos en tu aplicación. Todo excepto la Tabla se muestra cuando la búsqueda de datos está pendiente. Una vez que la solicitud resuelve el resultado, se muestra la tabla.

Ejercicios:

- leer más sobre React renderizado condicional[112]
- leer más sobre diferentes formas para renderizaciones condicionales[113]

[111] https://www.robinwieruch.de/conditional-rendering-react/
[112] https://facebook.github.io/react/docs/conditional-rendering.html
[113] https://www.robinwieruch.de/conditional-rendering-react/

Búsqueda por cliente o por servidor

Ahora cuando utilices el campo de búsqueda, filtrarás la lista. Sin embargo eso está sucediendo en el lado del cliente. Ahora vas a utilizar la API de Hacker News para buscar en el servidor. De lo contrario, sólo te ocuparías de la primera respuesta de la API que recibiste de `componentDidMount()` con el parámetro del término de búsqueda predeterminado.

Puede definir un método `onSubmit()` en su componente de clase ES6, que obtenga resultados de la API de Hacker News. Será la misma búsqueda que en tu método `componentDidMount()`. Pero lo buscara con el término modificado de la entrada del campo de búsqueda.

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result: null,
      searchTerm: DEFAULT_QUERY,
    };

    this.setSearchTopstories = this.setSearchTopstories.bind(this);
    this.fetchSearchTopstories = this.fetchSearchTopstories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...

  onSearchSubmit() {
    const { searchTerm } = this.state;
    this.fetchSearchTopstories(searchTerm);
  }

  ...
}
```

El componente de búsqueda obtiene un botón adicional. El botón tiene que activar explícitamente la búsqueda. De lo contrario, obtendrás datos de la API de Hacker News cada vez que tu campo de entrada cambie.

Como alternativa, podría rebatir (retrasar) la función `onChange()` y ahorrarte el botón, pero añadiría más complejidad en este momento. Vamos a mantenerlo simple sin un rebote.

Primero, pasa el método `onSearchSubmit()` a tu componente de búsqueda.

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
            onSubmit={this.onSearchSubmit}
          >
            Search
          </Search>
        </div>
        { result &&
          <Table
            list={result.hits}
            pattern={searchTerm}
            onDismiss={this.onDismiss}
          />
        }
      </div>
    );
  }
}
```

En segundo lugar, introduce un botón en tu componente de búsqueda. El botón tiene el `type="submit"` y el formulario utiliza su atributo `onSubmit()` para pasar el método `onSubmit()`. Puedes reutilizar la propiedad de los hijos, pero esta vez se utilizará como el contenido del botón.

```
const Search = ({
  value,
  onChange,
  onSubmit,
  children
}) =>
  <form onSubmit={onSubmit}>
    <input
      type="text"
      value={value}
      onChange={onChange}
    />
    <button type="submit">
      {children}
    </button>
  </form>
```

En la tabla puedes quitar la funcionalidad de filtro, ya que no habrá más ningún filtro en el cliente (búsqueda). El resultado viene directamente de la API Hacker News después de haber hecho clic en el botón “Buscar”.

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        ...
        { result &&
          <Table
            list={result.hits}
            onDismiss={this.onDismiss}
          />
        }
      </div>
    );
  }
}

...
```

```
const Table = ({ list, onDismiss }) =>
  <div className="table">
    { list.map(item =>
      ...
    )}
  </div>
```

Ahora cuando intentes buscar, notarás que el navegador se recarga. Es el comportamiento nativo del navegador para un callback en un formulario. En React, a menudo encontrarás el método de evento `preventDefault()` para suprimir el comportamiento nativo del navegador.

```
onSearchSubmit(event) {
  const { searchTerm } = this.state;
  this.fetchSearchTopstories(searchTerm);
  event.preventDefault();
}
```

Ahora deberías poder buscar diferentes historias de Hacker News. Interactúas con una API del mundo real. No debería haber más búsqueda de lado del cliente.

Ejercicios:

- leer mas sobre synthetic events in React[114]

[114] https://facebook.github.io/react/docs/events.html

Búsqueda paginada

¿Ha observado de cerca la estructura de datos devuelta? La Hacker News API[115] devuelve más que una lista de visitas. La propiedad de página, que es 0 en la primera respuesta, se puede utilizar para obtener más datos paginados. Sólo tiene que pasar la siguiente página con el mismo término de búsqueda a la API.

Vamos a extender las constantes API ensamblables para que pueda ocuparse de los datos paginados. const DEFAULT_QUERY = 'redux'; const DEFAULT_PAGE = 0;

const PATH_BASE = 'https://hn.algolia.com/api/v1'; const PATH_SEARCH = '/search'; const PARAM_SEARCH = 'query='; const PARAM_PAGE = 'page=';

Ahora puedes utilizar estas constantes para agregar el parámetro de página a tu solicitud de API.

```
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}`;

console.log(url);
// output: https://hn.algolia.com/api/v1/search?query=redux&page=
```

El método `fetchSearchTopstories()` tomará la página como segundo argumento. Los métodos `componentDidMount()` y `onSearchSubmit()` toman `DEFAULT_PAGE` para las llamadas iniciales a la API. Deben buscar la primera página en la primera solicitud. Cada búsqueda adicional debe buscar la siguiente página.

```
class App extends Component {

  ...

  componentDidMount() {
    const { searchTerm } = this.state;
    this.fetchSearchTopstories(searchTerm, DEFAULT_PAGE);
  }

  fetchSearchTopstories(searchTerm, page) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${pa\
ge}`)
      .then(response => response.json())
      .then(result => this.setSearchTopstories(result))
      .catch(e => e);
  }
```

[115] https://hn.algolia.com/api

```
  onSearchSubmit(event) {
    const { searchTerm } = this.state;
    this.fetchSearchTopstories(searchTerm, DEFAULT_PAGE);
    event.preventDefault();
  }

  ...

}
```

Ahora puede utilizar la página actual de la respuesta del API en `fetchSearchTopstories()`. Puedes utilizar este método en un botón para obtener más historias en un clic de botón. Utilicemos el botón para obtener más datos paginados de la API de Hacker News. Sólo necesitas definir la función `onClick()` que toma el término de búsqueda actual y la página siguiente (página actual + 1).

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    const page = (result && result.page) || 0;
    return (
      <div className="page">
        <div className="interactions">
        ...
        { result &&
          <Table
            list={result.hits}
            onDismiss={this.onDismiss}
          />
        }
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopstories(searchTerm, page + 1)}>
            More
          </Button>
        </div>
      </div>
    );
  }
}
```

Debe asegurarte de predeterminar a la página `0` cuando no hay ningún resultado.

Falta un paso. Traes la siguiente página de datos, pero sobrescribirás la página de datos anterior. Deseas concatenar los datos antiguos y nuevos. Vamos a ajustar la funcionalidad para agregar los nuevos datos en lugar de sobrescribirlos.

```
setSearchTopstories(result) {
  const { hits, page } = result;

  const oldHits = page !== 0
    ? this.state.result.hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    result: { hits: updatedHits, page }
  });
}
```

Primero, obtienes los hits y la página del resultado.

En segundo lugar, usted tiene que comprobar si ya hay antiguos hits. Cuando la página es `0`, es una nueva solicitud de búsqueda de `componentDidMount()` o `onSearchSubmit()`. . Los éxitos están vacíos. Pero cuando hace clic en el botón "More" para buscar datos paginados, la página no es `0`, está en la siguiente página. Es la página siguiente. Los hits antiguos ya están almacenados en tu estado y por lo tanto se pueden utilizar.

En tercer lugar, no deseas sobrescribir los antiguos resultados. Puedes combinar hits antiguos y nuevos de la solicitud de API reciente. La combinación de ambas listas se puede realizar con el operador de distribución de ES6 de JavaScript.

En cuarto lugar, establece los hits combinados y la página en el estado del componente interno.

Puedes hacer un último ajuste. Cuando pruebes el botón "More" sólo obtiene algunos elementos de la lista. El URL de la API se puede extender para buscar más elementos de la lista con cada solicitud. Una vez más, puede agregar más constantes de ruta compuestas.

```
const DEFAULT_QUERY = 'redux';
const DEFAULT_PAGE = 0;
const DEFAULT_HPP = '100';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';
const PARAM_HPP = 'hitsPerPage=';
```

Ahora puede usar las constantes para extender la URL de la API.

```
fetchSearchTopstories(searchTerm, page) {
  fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page\
}&${PARAM_HPP}${DEFAULT_HPP}`)
    .then(response => response.json())
    .then(result => this.setSearchTopstories(result))
    .catch(e => e);
}
```

Posteriormente, la solicitud a la Hacker News API busca más elementos de la lista en una solicitud que antes.

Ejercicios:

- experimentar con el Hacker News API parameters[116]

[116] https://hn.algolia.com/api

Caché del Cliente

Cada búsqueda envía una solicitud a la API de Hacker News. Puedes buscar “redux”, seguido de “react” y eventualmente “redux” de nuevo. En total hace 3 peticiones. Pero buscaste “redux” dos veces y las dos veces tomó todo un viaje de ida y vuelta asíncrono para buscar los datos. En una memoria caché del lado del cliente se almacenara cada resultado. Cuando se realiza una solicitud a la API, comprueba si ya hay un resultado. Si está allí, se utiliza la caché. De lo contrario, se realizará una solicitud de API para recuperar los datos.

Para tener un caché de cliente para cada resultado, tiene que almacenar múltiples `results` en lugar de un `result` en tu estado de componente interno. El objeto de resultados será un mapa con el término de búsqueda como clave y el resultado como valor. Cada resultado de la API se guardará mediante el término de búsqueda (clave).

En este momento tu resultado en el estado del componente es similar al siguiente:

```
result: {
  hits: [ ... ],
  page: 2,
}
```

Imagine que ha realizado dos solicitudes de API. Uno para el término de búsqueda “redux” y otro para “react”. El mapa de resultados debería tener el siguiente aspecto:

```
results: {
  redux: {
    hits: [ ... ],
    page: 2,
  },
  react: {
    hits: [ ... ],
    page: 1,
  },
  ...
}
```

Vamos a implementar una caché en el cliente con React `setState()`. En primer lugar, cambie el nombre del objeto `result` a `results` en el estado inicial del componente. En segundo lugar, defina una `searchKey` temporal que se utiliza para almacenar cada `result`.

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
    };

    ...

  }

  ...

}
```

La `searchKey` debe establecerse antes de realizar cada solicitud. Refleja el `searchTerm`. Usted puede preguntarse: ¿Por qué no usamos el `searchTerm` en primer lugar? El `searchTerm` es una variable fluctuante, ya que se cambia cada vez que escribes en el campo de búsqueda de entrada. Sin embargo, al final necesitarás una variable no fluctuante. Determina el término de búsqueda enviado recientemente a la API y puede utilizarse para recuperar el resultado correcto del mapa de resultados. Es un puntero a su resultado actual en la caché.

```
componentDidMount() {
  const { searchTerm } = this.state;
  this.setState({ searchKey: searchTerm });
  this.fetchSearchTopstories(searchTerm, DEFAULT_PAGE);
}

onSearchSubmit(event) {
  const { searchTerm } = this.state;
  this.setState({ searchKey: searchTerm });
  this.fetchSearchTopstories(searchTerm, DEFAULT_PAGE);
  event.preventDefault();
}
```

Ahora tiene que ajustar la funcionalidad donde el resultado se almacena en el estado del componente interno. Debe almacenar cada resultado por `searchKey`.

```
class App extends Component {

  ...

  setSearchTopstories(result) {
    const { hits, page } = result;
    const { searchKey, results } = this.state;

    const oldHits = results && results[searchKey]
      ? results[searchKey].hits
      : [];

    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      }
    });
  }

  ...

}
```

El `searchKey` se utilizará como clave para guardar los resultados y la página actualizados en un mapa de `results`.

Primero, tienes que recuperar el `searchKey` desde el estado del componente. Recuerde que e `searchKey` se pone en `componentDidMount()` y `onSearchSubmit()`.

En segundo lugar, los hits antiguos tienen que fusionarse con los nuevos hits como antes. Pero esta vez los antiguos hits son recuperados del mapa de `results` con el `searchKey` como clave.

En tercer lugar, un nuevo resultado puede establecerse en el mapa `results` en el estado. Examinemos el objeto `results` en `setState()`.

```
results: {
  ...results,
  [searchKey]: { hits: updatedHits, page }
}
```

La parte inferior se cerciora de almacenar el resultado actualizado por `searchKey` en el mapa de resultados. El valor es un objeto con una propiedad de visitas y página. El `searchKey` el término de búsqueda. Ya aprendiste la sintaxis `[searchKey]`. Es un nombre de propiedad computado ES6. Le ayuda a asignar valores dinámicamente en un objeto.

TLa parte superior tiene que objetar la propagación de todos los demás resultados por `searchKey` en el estado. De lo contrario, perdería todos los resultados almacenados anteriormente.

Ahora almacena todos los resultados por término de búsqueda. Ese es el primer paso para habilitar su caché. En el siguiente paso, puede recuperar el resultado dependiendo del término de búsqueda de su mapa de resultados.

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
      results[searchKey].hits
    ) || [];

    return (
      <div className="page">
        <div className="interactions">
          ...
```

```
        </div>
        <Table
          list={list}
          onDismiss={this.onDismiss}
        />
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopstories(searchKey, page + 1)}>
            More
          </Button>
        </div>
      </div>
    );
  }
}
```

Puesto que definiste a una lista vacía cuando no hay resultado por `searchKey`, Ahora puedes ahorrarte el renderizado condicional para el componente Tabla. Además, tendrá que pasar el `searchKey` en lugar del `searchTerm` al botón "More". De lo contrario, su búsqueda paginada depende del valor `searchTerm` que es fluctuante. Además, asegúrese de mantener la propiedad fluctuante `searchTerm` para el campo de entrada en el componente "Search".

La funcionalidad de búsqueda debería funcionar de nuevo. Almacena todos los resultados de la API de Hacker News.

Además, el método `onDismiss()` necesita ser mejorado. Todavía se ocupa del objeto `result`. Ahora tiene que tratar con múltiples `results`.

```
  onDismiss(id) {
    const { searchKey, results } = this.state;
    const { hits, page } = results[searchKey];

    const isNotId = item => item.objectID !== id;
    const updatedHits = hits.filter(isNotId);

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      }
    });
  }
```

El botón "Dismiss" debería funcionar de nuevo.

Sin embargo, nada impide que la aplicación envíe una solicitud de API en cada envío de búsqueda. Aunque puede haber ya un resultado, no hay ninguna comprobación que impida la solicitud. La funcionalidad de caché no está completa todavía. El último paso sería evitar la solicitud cuando un resultado está disponible en la caché.

```
class App extends Component {

  constructor(props) {

    ...

    this.needsToSearchTopstories = this.needsToSearchTopstories.bind(this);
    this.setSearchTopstories = this.setSearchTopstories.bind(this);
    this.fetchSearchTopstories = this.fetchSearchTopstories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  needsToSearchTopstories(searchTerm) {
    return !this.state.results[searchTerm];
  }

  ...

  onSearchSubmit(event) {
    const { searchTerm } = this.state;
    this.setState({ searchKey: searchTerm });

    if (this.needsToSearchTopstories(searchTerm)) {
      this.fetchSearchTopstories(searchTerm, DEFAULT_PAGE);
    }

    event.preventDefault();
  }

  ...

}
```

Ahora su cliente hace una solicitud a la API sólo una vez, aunque usted busca un término de búsqueda dos veces. Incluso los datos paginados con varias páginas se almacenan en caché de esa manera, porque siempre se guarda la última página de cada resultado en el mapa `results`.

¡Has aprendido a interactuar con una API en React! Repasemos los últimos capítulos:

- React
  - Métodos del ciclo de vida de los componentes de clase ES6 para diferentes casos de uso
  - componentDidMount() para las interacciones API
  - renderizados condicionales
  - eventos sintéticos en los formularios
- ES6
  - cadenas de plantilla para componer cadenas
  - operador de propagación para estructuras de datos inmutables
  - nombres de propiedad calculados
- General
  - Interacción con la API de Hacker News
  - búsqueda nativa con la API del navegador
  - búsqueda de cliente y servidor
  - paginación de datos
  - almacenamiento en caché del lado del cliente

Una vez más tiene sentido tomar un descanso. Internalizar los aprendizajes y aplicarlos por su cuenta. Puede experimentar con el código fuente que ha escrito hasta ahora.

Puedes encontrar el código fuente en el repositorio oficial[117].

[117] https://github.com/rwieruch/hackernews-client/tree/e60436a9d6c449e76a362aef44dd5667357b7994

Organización de Código y Pruebas

El capítulo se centrará en temas importantes para lograr que el código sea mantenible en una aplicación escalable. Aprenderás sobre la organización del código para adoptar las mejores prácticas al estructurar las carpetas y archivos. Otro aspecto que aprenderás son las pruebas, lo cual es importante para mantener un código robusto.

Módulos ES6: Importación y Exportación

En JavaScript ES6 puedes importar y exportar funcionalidades desde módulos. Estas funcionalidades pueden ser funciones, clases, componentes, constantes, etc. Básicamente todo lo que puede asignar a una variable. Los módulos pueden ser archivos individuales o carpetas enteras con un archivo de índice como punto de entrada.

Al principio del libro, después de haber iniciado la aplicación con *create-react-app* , existían varios `import` y `export` a través de los archivos iniciales. Ahora es el momento apropiado para explicar esto.

Las sentencias `import` y `export` ayudan a compartir código en varios archivos. Antes había varias soluciones para esto en el entorno JavaScript. Fue un desastre, porque no había una manera estándar para hacerlo, en la actualidad, es un comportamiento nativo de Javascript ES6.

Además, estas declaraciones adoptan la división de código. Se distribuye el código a través de varios archivos para mantenerlo reutilizable y mantenible. Lo primero es cierto porque puede importar el fragmento de código en varios archivos. Esto último es cierto porque usted tiene una sola fuente donde usted mantiene el pedazo de código.

Por último, pero no menos importante, le ayuda a pensar en encapsulación de código. No es necesario que todas las funcionalidades se exporten desde un archivo. Algunas de estas funcionalidades sólo deben utilizarse en el archivo donde se han definido. Las exportaciones de un archivo son básicamente la API pública del archivo. Sólo las funcionalidades exportadas están disponibles para ser reutilizadas en otro lugar. Sigue la mejor práctica de encapsulación.

Pero seamos prácticos. Como funcionan las declaraciones `import` y `export`? Los ejemplos siguientes muestran las declaraciones compartiendo una o varias variables entre dos archivos. Al final, el enfoque puede escalar a varios archivos y podría compartir más que simples variables.

Se puede exportar una o mas variables. Se llama una exportación con nombre.

```
const firstname = 'robin';
const lastname = 'wieruch';

export { firstname, lastname };
```

Y se importan en otro archivo con la ruta asociada a ese archivo:

```
import { firstname, lastname } from './file1.js';

console.log(firstname);
// output: robin
```

Tambien se pueden importar todas las variables asociadas con otro archivo:

```
import * as person from './file1.js';

console.log(person.firstname);
// output: robin
```

Todo lo que se importa puede tener un alias asociado. Esto es útil cuando en distintos archivos se exporta con el mismo nombre. Para solucionar el inconveniente de importación, se usa una alias:

```
import { firstname as foo } from './file1.js';

console.log(foo);
// output: robin
```

Por último pero no menos importante, existe la declaración `default`. Se puede utilizar para algunos casos de uso:

- exportar e importar una sola funcionalidad
- para resaltar la funcionalidad principal de la API exportada de un módulo
- tener una funcionalidad de importación alternativa

```
const robin = {
  firstname: 'robin',
  lastname: 'wieruch',
};

export default robin;
```

```
import developer from './file1.js';

console.log(developer);
// output: { firstname: 'robin', lastname: 'wieruch' }
```

El nombre de importación puede diferir del nombre predeterminado exportado. También puede utilizarlo junto con las instrucciones de exportación e importación nombradas.

```
const firstname = 'robin';
const lastname = 'wieruch';

const person = {
  firstname,
  lastname,
};

export {
  firstname,
  lastname,
};

export default person;


import developer, { firstname, lastname } from './file1.js';

console.log(developer);
// output: { firstname: 'robin', lastname: 'wieruch' }
console.log(firstname, lastname);
// output: robin wieruch
```

Se puede exportar directamente las variables:

```
export const firstname = 'robin';
export const lastname = 'wieruch';
```

Estas son las principales funcionalidades de los módulos ES6. Te ayudan a organizar tu código, a mantener tu código y a diseñar API de módulos reutilizables. También puedes exportar e importar funcionalidades para probarlas. Lo harás en uno de los siguientes capítulos.

Ejercicios:

- leer mas sobre ES6 import[118] - En español
- leer mas sobre ES6 export[119] - En español

[118] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/import
[119] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/export

Organización de código con módulos ES6

Podrías preguntarte: ¿Por qué no seguimos las mejores prácticas de división de código para el archivo *src/App.js*? En el fichero ya tenemos múltiples componentes que podrían definirse en sus propios ficheros/carpetas(módulos). Por el bien de aprender React, es práctico guardar estas cosas en un lugar. Pero una vez que una aplicación React crece, debes considerar dividir estos componentes en varios módulos. Sólo así la aplicación será escalable.

A continuación propongo varias estructuras de módulos que *podrías* aplicar. Yo recomendaría aplicarlos como un ejercicio al final del libro. Para mantener el libro en sí simple, no realizaré la división del código y seguiré los siguientes capítulos con el archivo *src/App.js*

Una posible estructura de módulo podría ser:

```
src/
  index.js
  index.css
  App.js
  App.test.js
  App.css
  Button.js
  Button.test.js
  Button.css
  Table.js
  Table.test.js
  Table.css
  Search.js
  Search.test.js
  Search.css
```

No parece muy prometedor. Puede ver un montón de nombres duplicados y sólo difiere la extensión de archivo. Otra estructura de módulo podría ser:

```
src/
  index.js
  index.css
  App/
    index.js
    test.js
    index.css
  Button/
    index.js
    test.js
```

```
    index.css
  Table/
    index.js
    test.js
    index.css
  Search/
    index.js
    test.js
    index.css
```

Parece más limpio que el anterior. Un componente se define por su declaración de componente en el archivo JavasScript, pero también por su estilo y pruebas.

Otro paso podría ser extraer las variables constantes del componente App. Estas constantes se utilizaron para componer el URL de la API de Hacker News.

```
src/
  index.js
  index.css
  constants/
    index.js
  components/
    App/
      index.js
      test.js
      index..css
    Button/
      index.js
      test.js
      index..css
    ...
```

Naturalmente, los módulos se dividirían en *src/constants/* y *src/components/*.

Ahora el archivo *src/constants/index.js* podría ser similar al siguiente:

```
export const DEFAULT_QUERY = 'redux';
export const DEFAULT_PAGE = 0;
export const DEFAULT_HPP = '100';

export const PATH_BASE = 'https://hn.algolia.com/api/v1';
export const PATH_SEARCH = '/search';
export const PARAM_SEARCH = 'query=';
export const PARAM_PAGE = 'page=';
export const PARAM_HPP = 'hitsPerPage=';
```

El archivo *App/index.js* podría importar estas variables para poder usarlas.

```
import {
  DEFAULT_QUERY,
  DEFAULT_PAGE,
  DEFAULT_HPP,

  PATH_BASE,
  PATH_SEARCH,
  PARAM_SEARCH,
  PARAM_PAGE,
  PARAM_HPP,
} from '../constants/index.js';

...
```

Cuando utilices la convención de nombrado de *index.js*, puedes omitir el nombre de archivo de la ruta relativa.

```
import {
  DEFAULT_QUERY,
  DEFAULT_PAGE,
  DEFAULT_HPP,

  PATH_BASE,
  PATH_SEARCH,
  PARAM_SEARCH,
  PARAM_PAGE,
  PARAM_HPP,
} from '../constants';

...
```

Pero ¿qué hay detrás del nombrado del archivo *index.js*? La convención se introdujo en el mundo node.js. El archivo de index es el punto de entrada a un módulo. Describe la API pública al módulo. Los módulos externos sólo pueden usar el archivo *index.js* para importar el código compartido del módulo. Considere la siguiente estructura de módulo compuesta para demostrarlo:

```
src/
  index.js
  App/
    index.js
  Buttons/
    index.js
    SubmitButton.js
    SaveButton.js
    CancelButton.js
```

La carpeta *Buttons/* tiene varios botones en forma de componentes definidos en archivos distintos. Cada archivo tiene su propio `export default` para brindar funcionalidad de exportación al componente y de esta manera, lo hace disponible para *Buttons/index.js*. El archivo general *Buttons/index.js* importa todos estos botones y las exporta publicamente.

```
import SubmitButton from './SubmitButton';
import SaveButton from './SaveButton';
import CancelButton from './CancelButton';

export {
  SubmitButton,
  SaveButton,
  CancelButton,
};
```

Ahora el *src/App/index.js* puede importar los botones de la API del módulo público ubicada en el archivo *index.js*.

```
import {
  SubmitButton,
  SaveButton,
  CancelButton
} from '../Buttons';
```

Seria una muy mala practica llegar a un determinado botón directamente sin pasar por *index.js*. Rompería las reglas de encapsulación.

```
// Mala practica, por favor no lo hagas.
import SubmitButton from '../Buttons/SubmitButton';
```

Ahora ya sabes cómo podrías refactorizar tu código fuente en módulos con las restricciones de la encapsulación. Como he dicho, por el bien de mantener el tutorial simple no voy a aplicar estos cambios. Pero debes hacer la refactorización al final del libro.

Ejercicios:

- refactorizar tu archivo *src/App.js* en módulos de múltiples componentes al finalizar el libro

Interfaz de componentes con PropTypes

Deberías conocer TypeScript[120] o Flow[121] para introducir una interfaz de tipo a JavaScript. Un lenguaje tipado es menos propenso a errores. Los editores y otras utilidades pueden detectar estos errores antes de que se ejecute el programa. Esto hace que su programa sea más robusto.

React viene con un comprobador de tipo incorporado para evitar errores. Puede utilizar PropTypes para describir la interfaz de componentes. Todas las props que se pasan de un componente principal a un componente secundario se validan basándose en la interfaz PropTypes asignada al componente secundario.

El capítulo mostrará cómo puedes hacer que todos los componentes sean seguros con PropTypes. Omitiré los cambios para los siguientes capítulos, porque agregan refactorings innecesarios de código. Pero debe mantenerlos y actualizarlos a lo largo del camino para mantener su tipo de interfaz de componentes seguro.

Inicialmente puede importar PropTypes. Tienes que ser cuidadoso de tu versión de React, porque en React versión 15.5 la importación cambió. Revisa el *package.json* para encontrar la versión de React.

Si es 15.5 o más, tiene que instalar un paquete independiente.

```
npm install --save prop-types
```

Si su versión es 15.4 o inferior, puede utilizar el paquete React ya instalado.

Ahora, dependiendo de su versión, puede importar PropTypes.

```
// React 15.5 y superior
import PropTypes from 'prop-types';

// React 15.4 e inferior
import React, { Component, PropTypes } from 'react';
```

Comencemos a asignar una interfaz de props a los componentes:

[120] https://www.typescriptlang.org/
[121] https://flowtype.org/

```
const Button = ({ onClick, className = '', children }) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

Button.propTypes = {
  onClick: PropTypes.func,
  className: PropTypes.string,
  children: PropTypes.node,
};
```

Eso es. Se toma cada argumento de la función y se le asigna un PropType. Los PropTypes básicos para primitivos y objetos complejos son:

```
* PropTypes.array
* PropTypes.bool
* PropTypes.func
* PropTypes.number
* PropTypes.object
* PropTypes.string
```

Además, tiene dos propTypes más para definir un fragmento renderizable (nodo), por ejemplo, una cadena y un elemento React.

```
* PropTypes.node
* PropTypes.element
```

Ya usaste el PropType `node` para el componente Button. En general hay más definiciones de PropType que puede leer en la documentación oficial de React.

At the moment all of the defined PropTypes for the Button are optional. The parameters can be null or undefined. But for several props you want to enforce that they are defined. You can make it a requirement that these props are passed to the component. Por el momento todos los propTypes definidos para el botón son opcionales. Los parámetros pueden ser nulos o no definidos. Pero para varios props deseas que se definan. Puedes hacer que sea un requisito que estas props se pasen al componente.

```
Button.propTypes = {
  onClick: PropTypes.func.isRequired,
  className: PropTypes.string,
  children: PropTypes.node.isRequired,
};
```

The `className` is not required, because it can default to an empty string. Next you will define a PropType interface for the Table component: El `className` no es necesario, ya que puede predeterminarse a una cadena vacía. A continuación, definirás una interfaz PropType para el componente Table:

```
Table.propTypes = {
  list: PropTypes.array.isRequired,
  onDismiss: PropTypes.func.isRequired,
};
```

Puedes definir el contenido de una matriz PropType más explícitamente:

```
Table.propTypes = {
  list: PropTypes.arrayOf(
    PropTypes.shape({
      objectID: PropTypes.string.isRequired,
      author: PropTypes.string,
      url: PropTypes.string,
      num_comments: PropTypes.number,
      points: PropTypes.number,
    })
  ).isRequired,
  onDismiss: PropTypes.func.isRequired,
};
```

Solo el `objectID` es requerido, porque tu sabes que parte de su código depende de ello. Las otras propiedades sólo se muestran, por lo que no son necesarias. Además, no puedes estar seguro de que la API de Hacker News siempre tenga una propiedad definida para cada objeto del array.

Eso es para PropTypes. Pero hay un aspecto más. Puede definir props predeterminadas en tu componente. Vamos a tomar de nuevo el componente Button. Las propiedades `className` tienen un parámetro predeterminado de ES6 en la firma de componente.

```
const Button = ({ onClick, className = '', children }) =>
  ...
```

Puede sustituirlo por el Prop de defecto interno de React:

```
const Button = ({ onClick, className, children }) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

Button.defaultProps = {
  className: '',
};
```

Igual que el parámetro predeterminado de ES6, el valor predeterminado garantiza que la propiedad se establece en un valor predeterminado cuando el componente primario no lo especifica. La comprobación de tipo PropType ocurre después de evaluar el valor predeterminado.

Ejercicios:

- respóndete las siguientes preguntas
- ¿el componente App tiene una interfaz PropType?
- definir la interfaz PropType para el componente Search
- agregue y actualice las interfaces PropType cuando agregue y actualice componentes en los próximos capítulos
- leer más sobre React PropTypes[122]

[122] https://facebook.github.io/react/docs/typechecking-with-proptypes.html

Pruebas instantáneas con Jest

Jest[123] es un framework JavaScript de pruebas. En Facebook se utiliza para validar el código JavaScript. En la comunidad React se utiliza para la cobertura de pruebas de componentes React . Por suerte *create-react-app* ya viene con Jest.

Comencemos a probar sus primeros componentes. Antes de que pueda hacer eso, tiene que exportar los componentes de su archivo *src/App.js* para testearlos en un archivo diferente.

```
...

class App extends Component {
  ...
}

...

export default App;

export {
  Button,
  Search,
  Table,
};
```

En tu archivo *App.test.js* encontrarás una primera prueba. Comprueba que el componente App se renderiza sin errores.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
});
```

Puedes ejecutarlo mediante el comando interactivo *create-react-app* en la línea de comandos. `npm run test`

Ahora Jest te permite escribir pruebas de instantáneas. Estas pruebas realizan una instantánea del componente representado y ejecutan esta instantánea contra futuras instantáneas. Cuando cambia

[123] https://facebook.github.io/jest/

una instantánea futura, se le notificará durante la prueba. Puedes aceptar el cambio de instantánea, ya que cambió la implementación del componente a propósito, o denegar el cambio e investigar por un error.

Jest almacena las instantáneas en una carpeta. Sólo de esa manera puede mostrar las diferencias a futuras instantáneas. Además, las instantáneas pueden compartirse entre equipos.

Debes instalar una biblioteca de utilidades antes de poder escribir su primera prueba de instantánea.

```
npm install --save-dev react-test-renderer
```

Ahora puede ampliar la prueba del componente App con su primera prueba instantánea.

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App from './App';

describe('App', () => {

  it('renders', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
  });

  test('snapshots', () => {
    const component = renderer.create(
      <App />
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Vuelva a ejecutar sus pruebas y verifica cómo las pruebas tienen éxito o fallan. Deben tener éxito. Una vez que cambie la salida del bloque de render en su componente App, la prueba de instantánea debe fallar. A continuación, puede decidir actualizar la instantánea o investigar en el componente de la aplicación.

Vamos a agregar más pruebas para nuestros componentes independientes. En primer lugar, el componente de búsqueda:

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App, { Search } from './App';
...

describe('Search', () => {

  it('renders', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Search>Search</Search>, div);
  });

  test('snapshots', () => {
    const component = renderer.create(
      <Search>Search</Search>
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

En segundo lugar, el componente Button:

```
...
import App, { Search, Button } from './App';
...

describe('Button', () => {

  it('renders', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Button>Give Me More</Button>, div);
  });

  test('snapshots', () => {
    const component = renderer.create(
      <Button>Give Me More</Button>
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
```

```
});
```

Por último, pero no menos importante, el componente Tabla:

```
...
import App, { Search, Button, Table } from './App';
...

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
  };

  it('renders', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Table { ...props } />, div);
  });

  test('snapshots', () => {
    const component = renderer.create(
      <Table { ...props } />
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Las pruebas instantáneas generalmente se mantienen bastante básicas. Sólo desea cubrir que el componente no cambia su salida. Una vez que cambia la salida, tiene que decidir si acepta los cambios. De lo contrario, tendrá que fijar el componente cuando la salida no sea la salida deseada.

Ejercicios:

- vea cómo fallan las pruebas de instantánea una vez que cambia la implementación de su componente
  - aceptar o denegar el cambio de instantánea

- mantén tus pruebas de instantáneas actualizadas cuando la implementación cambie en los próximos capítulos
- leer mas sobre Jest in React[124]

[124] https://facebook.github.io/jest/docs/tutorial-react.html

Pruebas unitarias con Enzyme

Enzyme[125] es una utilidad de prueba de Airbnb para afirmar, manipular y recorrer sus componentes React. Puede utilizarlo para realizar pruebas de unidad para complementar sus pruebas de instantánea.

Vamos a ver cómo se puede utilizar enzyme. Primero tienes que instalarlo ya que no viene con *create-react-app*.

```
npm install --save-dev enzyme react-addons-test-utils
```

Ahora puede escribir su primera prueba de unidad en el bloque de descripción de la tabla. Usaras `shallow()` para renderizar tu componente y asegurarte que la tabla tiene dos elementos.

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import { shallow } from 'enzyme';
import App, { Search, Button, Table } from './App';

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
  };

  ...

  it('shows two items in list', () => {
    const element = shallow(
      <Table { ...props } />
    );

    expect(element.find('.table-row').length).toBe(2);
  });

});
```

[125] https://github.com/airbnb/enzyme

Shallow renderiza el componentes sin componentes secundarios. Puedes hacer la prueba muy dedicada a un componente.

Enzyme tiene tres mecanismos de renderización en su API. Ya conoces `shallow()`, pero también existen `mount()` y `render()`. Ambos instancian instancias del componente principal y todos los componentes secundarios. Adicionalmente `mount()` te da más acceso a los métodos de ciclo de vida de los componentes. Pero, ¿cuándo utilizar qué mecanismo de renderización? Aquí algunas reglas básicas:

- Comience siempre con una prueba superficial `shallow()`
- Si `componentDidMount()` o `componentDidUpdate()` deben ser testeados, usa `mount()`
- Si deseas testear el ciclo de vida de los componentes y el comportamiento de los hijos, utiliza `mount()`
- Si deseas testear el renderizado de componentes hijos con menos gastos que `mount()` y no estás interesado en los métodos del ciclo de vida, use `render()`.

Usted podría continuar a la unidad de prueba de sus componentes. Pero asegúrese de mantener las pruebas simples y mantenibles. De lo contrario tendrá que refactorizarlos una vez que cambie sus componentes. Es por eso que Facebook introdujo las pruebas de Snapshot con Jest en primer lugar.

Ejercicios:

- mantenga sus pruebas de unidad actualizadas durante los siguientes capítulos
- leer más sobre enzyme and its rendering API[126]

¡Has aprendido cómo organizar tu código y cómo probarlo! Repasemos los últimos capítulos:

- React
  - PropTypes te permite definir controles de tipado de componentes
  - Jest le permite escribir pruebas instantáneas para sus componentes
  - Enzyme le permite escribir pruebas unitarias para sus componentes
- ES6
  - instrucciones de importación y exportación le ayudan a organizar su código
- General
  - La organización de código te permite escalar tu aplicación con las mejores prácticas

Puede encontrar el código fuente en e repositorio oficial[127].

[126] https://github.com/airbnb/enzyme
[127] https://github.com/rwieruch/hackernews-client/tree/393ce5a350aa34b1c7ae056333f7bb7b0807caef

Componentes React Avanzados

Este capítulo se enfoca en la implementación de componentes React avanzados. Aprenderás sobre componentes de orden superior y cómo implementarlos, navegando aún más profundo en el ecosistema React.

Ref a DOM Element

A veces necesitas interactuar con tus nodos DOM en React. El atributo `ref` te da acceso a un nodo dentro de tus elementos. Por lo general, ese se conoce como un antipatrón Reat, porque debes usar su forma declarativa de hacer las cosas y su flujo de datos unidireccional. Aprendiste sobre esto cuando se introdujo el primer campo de búsqueda, pero hay ciertos casos donde necesitas acceso al nodo DOM. La documentación oficial menciona tres de estos casos de uso:

- usar la API DOM (focus, media playback etc.)
- invocar animaciones de nodo DOM imperativas
- para integrar con una biblioteca de terceros que necesita el nodo DOM (por ejemplo D3.js[128])

Hagámoslo por ejemplo con el componente de búsqueda. Cuando la aplicación se renderiza por primera vez, el campo de entrada debe enfocarse. Ese es un caso de uso en el que sería necesario acceder a la API del DOM. En general, se puede utilizar el atributo `ref` tanto en componentes funcionales sin estado como en componentes de clase ES6. En este ejemplo, será necesario un método de ciclo de vida. Así que el procedimiento es propuesto utilizando el atributo `ref` con un componente de clase ES6.

El paso inicial es refactorizar el componente funcional sin estado (functional stateless component) a un componente de clase ES6.

src/App.js

```
class Search extends Component {
  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
        />
        <button type="submit">
          {children}
```

[128] https://d3js.org/

```
        </button>
      </form>
    );
  }
}
```

El objeto `this` de un componente de clase ES6 sirve para hacer referencia al nodo DOM con el atributo `ref`.

src/App.js

```
class Search extends Component {
  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
          ref={el => this.input = el}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

Ahora puedes enfocar el campo de entrada cuando el componente se monte usando el objeto `this`, el método del ciclo de vida apropiado y la API del DOM.

src/App.js

```
class Search extends Component {
  componentDidMount() {
    if (this.input) {
      this.input.focus();
    }
  }

  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
          ref={el => this.input = el}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

El campo de entrada debe enfocarse al momento en que la aplicación es renderizada. Pero, ¿cómo obtendrías acceso a `ref` en un componente funcional sin estado, sin el objeto `this`? El siguiente componente funcional sin estado lo demuestra.

src/App.js

```
const Search = ({
  value,
  onChange,
  onSubmit,
  children
}) => {
  let input;
  return (
    <form onSubmit={onSubmit}>
      <input
        type="text"
        value={value}
        onChange={onChange}
        ref={el => this.input = el}
      />
      <button type="submit">
        {children}
      </button>
    </form>
  );
}
```

Ahora se puede utilizar el elemento de entrada (`input`) del DOM. En el ejemplo del caso de uso de foco, esto no ayudaría mucho, porque no existe un método de ciclo de vida para desencadenar el enfoque. Así que por ahora no se utilizara la variable de entrada (`input`). Pero en el futuro es posible que encuentres otros casos de uso donde si tiene sentido utilizar un componente funcional sin estado con el atributo `ref`.

Ejercicios

- lee más sobre uso general del atributo ref en React[129]
- lee más sobre usos del atributo ref en React[130]

[129] https://facebook.github.io/react/docs/refs-and-the-dom.html
[130] https://www.robinwieruch.de/react-ref-attribute-dom-node/

Cargando ...

Ahora regresemos a la aplicación, donde posiblemente quieras mostrar un indicador de carga al enviar una solicitud de búsqueda a la API de Hacker News. La solicitud es asincrónica así que es necesario mostrarle al usuario algún indicador de que algo está sucediendo. Definamos un componente de carga reutilizable dentro del archivo *src/App.js*.

src/App.js

```
const Loading = () =>
  <div>Loading ...</div>
```

Ahora, será necesario agregar una propiedad para almacenar el estado de carga. En función del estado de carga, puede decidir mostrar el componente `Loading` más adelante.

src/App.js

```
class App extends Component {
  _isMounted = false;

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
      isLoading: false,
    };

    ...
  }

  ...

}
```

El valor inicial de la propiedad `isLoading` es falso. No se carga nada antes de montar el componente de la aplicación.

Una vez la solicitud es realizada, establece un estado de la propiedad cambia a verdadero. Finalmente, la solicitud tendrá éxito y puede establecer el estado de carga en falso.

src/App.js

```
class App extends Component {

  ...

  setSearchTopStories(result) {
    ...

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      },
      isLoading: false
    });
  }

  fetchSearchTopStories(searchTerm, page = 0) {
    this.setState({ isLoading: true });

    axios(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${pa\
ge}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(result => this._isMounted && this.setSearchTopStories(result.data))
      .catch(error => this._isMounted && this.setState({ error }));
  }

  ...

}
```

En el último paso, usarás el componente `Loading`dentro del componente `App`. Una representación condicional basada en el estado de carga decidirá si muestra el componente `Loading` o el componente `Button`. El último es el botón para obtener más datos.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <div className="interactions">
          { isLoading
            ? <Loading />
            : <Button
                onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}
              >
              More
            </Button>
          }
        </div>
      </div>
    );
  }
}
```

Inicialmente, el componente `Loading` aparecerá al iniciar la aplicación, ya que realiza una solicitud en `componentDidMount()`. No hay un componente `Table` porque la lista está vacía. Cuando se obtiene la respuesta de la API Hacker News, se muestra el resultado, el estado de carga se establece en falso y el componente `Loading` desaparece. Entonces, aparece el botón "More" para buscar más datos. Una vez que se obtengan más datos, el botón desaparecerá. De lo contrario, se mostrará el componente `Loading`.

Ejercicios:

- usa una librería como Font Awesome[131] para mostrar un icono de carga en lugar del texto "Loading ..."

[131] http://fontawesome.io/

Componentes de orden superior (Higher-Order Components)

Componentes de orden superior (HOC) son un concepto avanzado en React. HOCs son equivalentes a las funciones de orden superior. Toman cualquier entrada, la mayoría de las veces un componente, pero también argumentos opcionales, y devuelven un componente como resultado. El componente devuelto es una versión mejorada del componente de entrada y se puede usar con JSX.

Los HOCs se utilizan en diferentes casos de uso. Pueden preparar propiedades, gestionar el estado o alterar la representación de un componente. Un caso de uso podría ser utilizar un HOC como ayudante para una representación condicional. Imagina que tienes un componente `List` que muestra una lista de elementos o nada, porque la lista está vacía o nula. El HOC podría ocultar que la lista no representaría nada cuando no hay una lista. Por otro lado, el componente `List` simple no necesita preocuparse por una lista inexistente. Solo le importa renderizar la lista.

Hagamos una HOC simple que tome un componente como entrada y devuelva otro componente. Puedes colocarlo en tu archivo *src/App.js*.

src/App.js

```
function withFoo(Component) {
  return function(props) {
    return <Component { ...props } />;
  }
}
```

Una útil convención es prefijar el nombre de un HOC con `with` . Dado que está utilizando JavaScript ES6, puedes expresar el HOC de manera más concisa con una función de flecha ES6.

src/App.js

```
const withEnhancement = (Component) => (props) =>
  <Component { ...props } />
```

En el ejemplo anterior, el componente de entrada se mantendría igual que el componente de salida. No pasa nada. El componente de salida debería mostrar el componente `Loading` tan pronto el estado cambie a verdadero, de otra manera debería mostrar el componente `Input`

src/App.js

```
const withLoading = (Component) => (props) =>
  props.isLoading
    ? <Loading />
    : <Component { ...props } />
```

Según la propiedad de carga, puede aplicar una renderización condicional. La función devolverá el componente `Loading` o el componente `Input`.

En general, puede ser bastante eficiente distribuir un objeto, como el objeto `props`, como entrada para un componente. En el siguiente fragmento de código se puede ver la diferencia:

Code Playground

```
// before you would have to destructure the props before passing them
const { firstname, lastname } = props;
<UserProfile firstname={firstname} lastname={lastname} />

// but you can use the object spread operator to pass all object properties
<UserProfile { ...props } />
```

Hay una pequeña cosa que debes evitar. Se pasaron todas las `props`, incluyendo la propiedad `isLoading` al pasar el objeto al componente `Input`. Sin embargo, el componente `Input` no no la existencia de la propiedad `isLoading`. Puedes usar la re-desestructuración de ES6 para evitar esto.

src/App.js

```
const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />
```

Se toma una propiedad del objeto, pero se conserva el resto del objeto. También funciona con múltiples propiedades. Puedes leer mas acerca de esto en el articulo de la página de Mozilla: destructuring assignment[132].

Ahora puedes usar HOCs dentro de JSX. Un caso de uso en la aplicación podría ser mostrar el botón "More" o el componente `Loading`. El componente `Loading` ya está encapsulado dentro del HOC, pero falta un componente `Input`. Para mostrar un componente `Button` o bin un componente `Loading`, el Button es el componente de entrada del HOC. El componente de salida mejorado es un componente `ButtonWithLoading`.

[132] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

src/App.js

```
const Button = ({
  onClick,
  className = '',
  children,
}) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

const Loading = () =>
  <div>Loading ...</div>

const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />

const ButtonWithLoading = withLoading(Button);
```

Todo se encuentra definido ya. Como último paso, debes usar el componente `ButtonWithLoading`, que recibe el estado de carga como una propiedad adicional. Mientras el HOC consume la propiedad de carga, todas las demás propiedad pasan al componente `Button`.

src/App.js

```
class App extends Component {

  ...

  render() {
    ...
    return (
      <div className="page">
        ...
        <div className="interactions">
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}
```

```
          >
            More
          </ButtonWithLoading>
        </div>
      </div>
    );
  }
}
```

Cuando ejecute nuevamente las pruebas, notarás que la prueba para el componente `App` falla. La línea de comando debería mostrar algo como esto:

Command Line

```
-     <button
-       className=""
-       onClick={[Function]}
-       type="button"
-     >
-       More
-     </button>
+     <div>
+       Loading ...
+     </div>
```

Ahora, puedes intentar arreglar el componente al momento en que pienses que algo está mal con el, o puedes aceptar este pequeño error. Y como en este capítulo acabas de introducir al componente `Loading`, puedes aceptar el pequeño error mostrado en la línea de comandos al realizar la prueba interactiva.

Componentes de orden superior son un partón avanzado en React. Tienen multiples propósitos: Mejorada reusabilidad de componentes, mayor abstracción, composibilidad de componentes y la manipulación de props, estados y vistas. Puedes leer gentle introduction to higher-order components[133]. Te ofrece otro enfoque para aprender sobre este tema, te ofrece una manera elegante de de usarlos dentro del paradigma de programación funcional y resuelve el problema de renderizado condicional con componentes de orden superior.

Exercises:

- Lee a gentle introduction to higher-order components[134]
- Experimenta con el HOC que acabas de crear
- Piensa sobre un caso de uso donde otro HOC tendría sentido
- Implementa el HOC, si hay un caso de uso para ello

[133] https://www.robinwieruch.de/gentle-introduction-higher-order-components/
[134] https://www.robinwieruch.de/gentle-introduction-higher-order-components/

Sorting Avanzado

Ya implementaste un campo de búsqueda que interactúa con el lado del servidor y el lado de la aplicación. Y como tienes un componente `Table`, tiene sentido intentar mejorarlo con interacciones un poco mas complejas. Acto seguido, agregarás una función para organizar cada columna utilizando el encabezado de la Tabla.

Es posible que escribas tu propia función de clasificación de elementos en la tabla, pero prefiero usar una utilidad incluida en librerías como Lodash[135]. Hay otras opciones, pero en el libro usaremos Lodash.

Command Line

```
npm install lodash
```

Ahora, es posible importar la función de clasificación incluida en Lodash dentro del archivo *src/App.js*:

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import { sortBy } from 'lodash';
import './App.css';
```

Ya tienes varias columnas en la tabla: title, author, comments y points. Puedes definir funciones de clasificación (sort functions) donde cada una toma una lista y returna una lista de elementos clasificados de acuerdo a una propiedad específica. Adicionalmente, necesitarás una función de clasificación por defecto que no clasifique, pero que retorne una lista no clasificada. Esta lista será el estado inicial.

src/App.js

```
...

const SORTS = {
  NONE: list => list,
  TITLE: list => sortBy(list, 'title'),
  AUTHOR: list => sortBy(list, 'author'),
  COMMENTS: list => sortBy(list, 'num_comments').reverse(),
  POINTS: list => sortBy(list, 'points').reverse(),
};
```

[135] https://lodash.com/

```
class App extends Component {
  ...
}
...
```

Dos de las funciones de clasificación retornan una lista invertida. Es para ver los ítems con la mayor cantidad de números y puntos, en vez de los ítems con el menor conteo al momento en que la lista es clasificada por primera vez.

Los objetos `SORTS` permiten referenciar a cualquier función de clasificación a partir de ahora.

El componente `App` es responsable de almacenar el estado de la clasificación. El estado inicial será la opción de clasificación por defecto, que no clasifica nada en absoluto, solo retorna la lista.

src/App.js

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  error: null,
  isLoading: false,
  sortKey: 'NONE',
};
```

Una vez que has una `sortKey` diferente, por ejemplo `AUTHOR`, es posible clasificar la lista con la función de clasificación adecuada.

Ahora, podemos definir un nuevo método dentro del componente `App` que inicializa un `sortKey` para el estado local del componente, la `sortKey` puede ser utilizada para para ayudar a aplicar la función de clasificación correcta a la lista:

src/App.js

```
class App extends Component {
  _isMounted = false;

  constructor(props) {

    ...

    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
```

```
    this.onDismiss = this.onDismiss.bind(this);
    this.onSort = this.onSort.bind(this);
  }

  ...

  onSort(sortKey) {
    this.setState({ sortKey });
  }

  ...

}
```

El siguiente paso consiste en pasar el método y el `sortKey` al componente `Tabla`.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading,
      sortKey
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <Table
          list={list}
          sortKey={sortKey}
          onSort={this.onSort}
          onDismiss={this.onDismiss}
        />
        ...
```

```
      </div>
    );
  }
}
```

El componente `Table` es responsable de clasificar la lista. Este componente toma una de las funciones `SORT` y le pasa la lista como entrada, y luego sigue mapeando la lista previamente clasificada.

src/App.js

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
  <div className="table">
    {SORTS[sortKey](list).map(item =>
      <div key={item.objectID} className="table-row">
        ...
      </div>
    )}
  </div>
```

En teoría, la lista debería se clasificada por una de las funciones. Sin embargo la clasificación por defecto está configurada con el valor `NONE`, así que nada es clasificado aún, porque nada ejecuta al método `onSort()` para cambiar el valor de `sortKey`. Se extiende la Tabla con una fila de encabezados de columna que utilizan componentes `Sort` dentro de las columnas para realizar la clasificación de cada una de las columnas:

src/App.js

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
  <div className="table">
    <div className="table-header">
      <span style={{ width: '40%' }}>
        <Sort
          sortKey={'TITLE'}
          onSort={onSort}
```

```
        >
          Title
        </Sort>
      </span>
      <span style={{ width: '30%' }}>
        <Sort
          sortKey={'AUTHOR'}
          onSort={onSort}
        >
          Author
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        <Sort
          sortKey={'COMMENTS'}
          onSort={onSort}
        >
          Comments
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        <Sort
          sortKey={'POINTS'}
          onSort={onSort}
        >
          Points
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        Archive
      </span>
    </div>
    {SORTS[sortKey](list).map(item =>
      ...
    )}
  </div>
```

Cada componente `Sort` recibe una `sortKey` especifica y la función general `onSort()`. Internamente llama al método con la `sortKey` correspondiente, para determinar la `sortKey` específica.

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button onClick={() => onSort(sortKey)}>
    {children}
  </Button>
```

Como puedes ver, el componente `Sort` reutiliza el componente `Button`. Durante el click del botón, cada `sortKey` será asignada de manera individual por el método `onSort()`. Ahora será posible clasificar la lista cuándo el encabezado de la columna este seleccionado.

Ahora, mejorarás el estilo del botón en el encabezado de cada columna. Dale un `className` adecuado:

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button
    onClick={() => onSort(sortKey)}
    className="button-inline"
  >
    {children}
  </Button>
```

Esto se hace con el fin de mejorar la UI (Interfaz de Usuario por sus siglas en Inglés). El siguiente objetivo es implementar una clasificación invertida (reverse sort). La lista debería realizar una clasificación invertida al momento en que un componente `Sort` es clickeado dos veces. En primer lugar, es necesario que definas el estado reverso con un booleano. La clasificación puede ser invertida o no invertida (reversed ó non-reversed).

src/App.js

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  error: null,
  isLoading: false,
  sortKey: 'NONE',
  isSortReverse: false,
};
```

Ahora dentro del método `Sort`, es posible evaluar si la lista está sorteada en reverso. Se determina que está en reverso si el estado es el mismo que el de la `sortKey` nueva y el estado aún no posee un valor igual a verdadero (true).

src/App.js

```
onSort(sortKey) {
  const isSortReverse = this.state.sortKey === sortKey && !this.state.isSortReverse;
  this.setState({ sortKey, isSortReverse });
}
```

De nuevo, es posible pasar la propiedad revertida al componente `Tabla`:

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading,
      sortKey,
      isSortReverse
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <Table
          list={list}
          sortKey={sortKey}
          isSortReverse={isSortReverse}
          onSort={this.onSort}
          onDismiss={this.onDismiss}
        />
        ...
      </div>
    );
  }
}
```

La Tabla debe tener una bloque correspondiente a la función flecha con la capacidad de computar datos:

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return(
    <div className="table">
      <div className="table-header">
        ...
      </div>
      {reverseSortedList.map(item =>
        ...
      )}
    </div>
  );
}
```

Finalmente, queremos brindar información visual al usuario para que pueda distinguir cuáles columnas están clasificadas de manera activa. Cada componente `Sort` ya tiene su `sortKey` específico, el cuál puede ser utilizado para identificar cuando la clasificación esta activa o no. Se pasa el `sortKey` desde el estado del componente interno como una clasificación activa al componente `Sort`:

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
```

```
    ? sortedList.reverse()
    : sortedList;

  return(
    <div className="table">
      <div className="table-header">
        <span style={{ width: '40%' }}>
          <Sort
            sortKey={'TITLE'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Title
          </Sort>
        </span>
        <span style={{ width: '30%' }}>
          <Sort
            sortKey={'AUTHOR'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Author
          </Sort>
        </span>
        <span style={{ width: '10%' }}>
          <Sort
            sortKey={'COMMENTS'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Comments
          </Sort>
        </span>
        <span style={{ width: '10%' }}>
          <Sort
            sortKey={'POINTS'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Points
          </Sort>
        </span>
        <span style={{ width: '10%' }}>
```

```
          Archive
        </span>
      </div>
      {reverseSortedList.map(item =>
          ...
      )}
    </div>
  );
}
```

Dentro del componente `Sort`, se puede saber si la función de clasificación esta activa en base a `sortKey` y `activeSortKey`. Al componente `Sort` dale un atributo extra llamado `className`, para brindarle información acerca de su estado al usuario:

src/App.js

```
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = ['button-inline'];

  if (sortKey === activeSortKey) {
    sortClass.push('button-active');
  }

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass.join(' ')}
    >
      {children}
    </Button>
  );
}
```

Es posible definir `sortClass` de manera más eficiente utilizando la librería llamada `classnames`, que se instala utilizando npm:

Command Line

```
npm install classnames
```

Después de la instalación, lo importamos en el tope del archivo *src/App.js*.

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import { sortBy } from 'lodash';
import classNames from 'classnames';
import './App.css';
```

Ahora, ya puedes usarlo para definir el componente `className` con clases condicionales.

src/App.js

```
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = classNames(
    'button-inline',
    { 'button-active': sortKey === activeSortKey }
  );

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass}
    >
      {children}
    </Button>
  );
}
```

Habrán unidades de prueba fallidas en el componente `Table`. Dado que cambiamos intencionalmente la representación de nuestro componente, aceptamos los resultados de las pruebas, pero todavía es necesario repara la unidad de prueba. En *src/app.test.js*, ingresa una función `sortKey` y el booleano `isSortReverse` para el componente `Tabla`.

src/App.test.js

```
...

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
    sortKey: 'TITLE',
    isSortReverse: false,
  };

  ...

});
```

De nuevo, es necesario aceptar las fallas en las pruebas para el componente `Table`, porque utilizamos props extendidas en el. La interacción de clasificación avanzada finalmente ha sido completada.

Ejercicios:

- Usa una librería como Font Awesome[136] para indicar la clasificación (reverse). Podrías utilizar una flecha hacia arriba o una flecha hacia abajo al lado de cada Encabezado
- Lee más acerca de classnames library[137]

[136] http://fontawesome.com/
[137] https://github.com/JedWatson/classnames

Aprendiste sobre técnicas de componentes avanzados en React. Recapitulemos lo visto en el capítulo:

- **React**
  - El atributo `ref` hace referencia a elementos que forman parte del DOM
  - Componentes de orden superior son comúnmente utilizados para construir componentes avanzados
  - Implementación de interacciones avanzadas en React
  - `classNames` condicionales utilizando una librería
- **ES6**
  - Desestructuración para dividir objetos y arrays

Puedes encontrar el código fuente en el repositorio oficial[138].

[138] https://github.com/rwieruch/hackernews-client/tree/9456117fb67bbe98d7e3f41bbc85b4a035020e7e

Manejo del Estado en React y mucho más

Conociste en capítulos previos los conceptos básicos del manejo del estado en React. Este capítulo profundiza un poco más en el tema. Aprenderás buenas prácticas, como aplicarlas y porque podrías considerar usar una librería especialmente diseñada para la gestión de estado.

Traspaso del Estado

El único componente con estado en tu aplicación, usando ES6, es el componente `App`. Se encarga de gran parte del estado y lógica (métodos) de la aplicación. Quizás hayas notado que pasaste muchas propiedades a tu componente `Table`. De las cuáles, la mayoría de sólo se usan dentro de dicho componente. No tiene sentido que el componente `App` reconozca estas propiedades.

Por ejemplo, la función de clasificación solamente se utiliza en el componente `Table`. Podrías mover esta funcionalidad adentro del componente. Ya que el componente `App` no necesita saber nada de ella.

El proceso de refactorización de un subestado desde un componente a otro es conocido como *traspaso del estado* (lifting state). En tu caso, quieres mover el estado que no es usado en el componente `App`, más cerca del componente `Table`. El estado pasa del componente padre al componente hijo.

Para tratar con los estados y métodos en el componente `Table`, este tiene que convertirse a un componente de clase usando ES6. La refactorización de componente funcional sin estado a un componente de clase usando ES6 puede lograrse de manera sencilla.

El componente `Table` como un componente funcional sin estado:

Your Table component as a functional stateless component:

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return(
    ...
  );
}
```

Puede refactorizarse a un componente de clase usando ES6:

src/App.js

```
class Table extends Component {
  render() {
    const {
      list,
      sortKey,
      isSortReverse,
      onSort,
      onDismiss
    } = this.props;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return (
      ...
    );
  }
}
```

Ya que quieres trabajar con estados y métodos en tu componente, tienes que añadir un constructor y un estado inicial:

src/App.js

```
class Table extends Component {
  constructor(props) {
    super(props);

    this.state = {};
  }

  render() {
    ...
  }
}
```

Ahora, puedes mover estados y métodos de clase con la funcionalidad de clasificación desde el componente `App` hasta el componente `Table`.

src/App.js

```
class Table extends Component {
  constructor(props) {
    super(props);

    this.state = {
      sortKey: 'NONE',
      isSortReverse: false,
    };

    this.onSort = this.onSort.bind(this);
  }

  onSort(sortKey) {
    const isSortReverse = this.state.sortKey === sortKey &&
      !this.state.isSortReverse;

    this.setState({ sortKey, isSortReverse });
  }

  render() {
    ...
  }
}
```

Recuerda que es necesario borrar el estado y el método de clase que moviste desde tu componente `App`

src/App.js

```
class App extends Component {
  _isMounted = false;

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
      isLoading: false,
    };
```

```
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
  }

  ...

}
```

También puedes hacer mas ligero al componente `Table`. Para ello, debes mover las propiedades que le son asignadas a este y que se encuentran dentro del componente `app`, porque son utilizadas de manera interna por el componente `Table`.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        { error
          ? <div className="interactions">
            <p>Something went wrong.</p>
          </div>
          : <Table
            list={list}
            onDismiss={this.onDismiss}
```

```
          />
        }
        ...
      </div>
    );
  }
}
```

Dentro del componente `Table`, usa el método interno `onSort()` y el estado interno del componente `Table`:

src/App.js

```
class Table extends Component {

  ...

  render() {
    const {
      list,
      onDismiss
    } = this.props;

    const {
      sortKey,
      isSortReverse,
    } = this.state;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return(
      <div className="table">
        <div className="table-header">
          <span style={{ width: '40%' }}>
            <Sort
              sortKey={'TITLE'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Title
            </Sort>
```

```
          </span>
          <span style={{ width: '30%' }}>
            <Sort
              sortKey={'AUTHOR'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Author
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'COMMENTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Comments
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'POINTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Points
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            Archive
          </span>
        </div>
        { reverseSortedList.map((item) =>
          ...
        )}
      </div>
    );
  }
}
```

Acabas de realizar una refactorización crucial al mover la funcionalidad y el estado de un componente a otro componente para aligerar al componente del que se extraen los métodos y estado.

De nuevo, el componente `API` perteneciente a `Table` se volvió más ligero, porque internamente, este interactúa con la funcionalidad de clasificación.

El traspaso de estado puede realizarse de otra manera también: De componente hijo a componente padre. Esto se conoce como traspaso inverso de estado (`lifting state up`). Imagina que estas trabajando con el estado local dentro de un componente hijo y quieres que el estado sea visible dentro del componente padre también. Sería necesario que traspases el estado al componente padre. Es decir, realizar un traspaso de estado invertido de estado. En este caso, el componente padre lidia con el estado local y permite que este sea accesible para todos sus componentes hijo.

Ejercicios:

- Lee acerca de traspaso de estado en React[139]
- Lee acerca de el traspaso de estado en el artículo learn React before using Redux[140]

[139] https://reactjs.org/docs/lifting-state-up.html
[140] https://www.robinwieruch.de/learn-react-before-using-redux/

Revisión: setState()

Hasta ahora, has utilizado la función `setState()` de React para gestionar el estado interno de los componentes. Es posible pasar un objeto a la función donde se actualize parte del estado local.

Code Playground

```
this.setState({ value: 'hello'});
```

Pero, `setState()` no toma solamente un objeto. En su segunda versión, puedes pasar una función para actualizar el estado.

Code Playground

```
this.setState((prevState, props) => {
  ...
});
```

Hay un caso crucial donde tiene sentido utilizar una función en vez de un objeto: Cuando actualizas el estado dependiendo del estado previo o de las propiedades. Si no utilizas una función, la gestión local de estado puede generar fallas o bugs. El método `setState()` de React es asíncrono. React recopila todas las llamadas `setState()` y las ejecuta eventualmente. Algunas veces, el estado anterior cambia antes de la llamada al método `setStare()`.

Code Playground

```
const { oneCount } = this.state;
const { anotherCount } = this.props;
this.setState({ count: oneCount + anotherCount });
```

Imagina que `oneCount` y `anotherCount` cambian de manera asíncrona en alguna otra parte del código al momento en que llamas a `setState()`. En una aplicación en constante crecimiento, habrán varias llamadas a `setState()` a lo largo de dicha aplicación. Gracias a que `setState()` se ejecuta de forma asíncrona, para este caso podrías utilizar valores antiguos.

Con el enfoque en funciones, la función en `setState()` es un callback que opera en el estado y propiedades al tiempo en que se ejecuta la función callback. Incluso cuándo `setState()` es asíncrono, con una función se toman el estado y las propiedades al mismo tiempo en que `setState()` se ejecuta.

Code Playground

```
this.setState((prevState, props) => {
  const { oneCount } = prevState;
  const { anotherCount } = props;
  return { count: oneCount + anotherCount };
});
```

En nuestro código, el método `setSearchTopStories()` utiliza el estado anterior y este es un buen ejemplo de cuándo utilizar una función en vez de un objeto dentro de `setState()`. Justo ahora, el código luce así:

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;
  const { searchKey, results } = this.state;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    results: {
      ...results,
      [searchKey]: { hits: updatedHits, page }
    },
    isLoading: false
  });
}
```

Aquí se extraen los valores del estado, pero el estado se actualizó de forma asíncrona en base al estado anterior. Ahora, usaremos el enfoque funcional para prevenir bugs originados por estados antiguos:

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
    ...
  });
}
```

Hace un momento implementamos un bloque que ahora puedes mover dentro de la función redireccionándolo para que opere en `prevState` en vez de `this.state`.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
    const { searchKey, results } = prevState;

    const oldHits = results && results[searchKey]
      ? results[searchKey].hits
      : [];

    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    return {
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      },
      isLoading: false
    };
  });
}
```

Esto resolverá la situación con un estado obsoleto, pero todavía hay una mejoría que puede implementarse. Debido a que es una función, puede extraerla y así mejorar la legibilidad del código. Una ventaja extra de utilizar una función en vez de un objeto es que la función puede estar ubicada fuera del componente. Aún sería necesario utilizar una función de alto nivel para para pasarle el resultado porque queremos actualizar el estado en base al resultado obtenido de la API.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;
  this.setState(updateSearchTopStoriesState(hits, page));
}
```

La función `updateSearchTopStoriesState()` tiene que retornar una función. Es una función de alto nivel (`higher-order function`) que puede ser definida fuera del componente `App`. Nota como la función personalizada cambia ligeramente ahora.

src/App.js

```
const updateSearchTopStoriesState = (hits, page) => (prevState) => {
  const { searchKey, results } = prevState;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  return {
    results: {
      ...results,
      [searchKey]: { hits: updatedHits, page }
    },
    isLoading: false
  };
};

class App extends Component {
  ...
}
```

El enfoque en funciones y no en objetos previene posibles fallas o bugs dentro de `setState()`, a la vez que la legibilidad y la mantenibilidad del código también se ven mejoradas. Además, se pueden realizar pruebas fuera del componente `App`. Recomiendo el exportar y realizar pruebas como una buena practica.

Ejercicios:

- Lee sobre como utilizar de manera correcta el estado en React[141]
- Exporta updateSearchTopStoriesState desde el archivo
  - Escribe una prueba que pase `hits`, `page` y un estado previo para que finalmente se espere un estado nuevo.
- Refactoriza tus métodos `setState()` para que utilicen una función, si consideras que es necesario porque utiliza propiedades o estados
- Ejecuta de nuevo tus pruebas y verifica que todo esté actualizado

[141] https://reactjs.org/docs/state-and-lifecycle.html#using-state-correctly

Domando el Estado

Capítulos anteriores te enseñaron que la gestión de estado puede llegar a ser un tema crucial en aplicaciones grandes, así como React y otros frameworks SPA tienen problemas con esto. A medida que las aplicaciones se hacen mas y más complejas, el gran reto en las aplicaciones web es el de domar y controlar el estado.

En comparación con otras soluciones, React ya ha tomado un paso a la delantera. Un flujo de datos unidireccional y una API simple que sirvan para gestionar estados en los componentes es indispensable. Estos conceptos hacen mas sencillo el lidiar con los estados y los cambios de estado. También facilita lidiar con ellos a nivel de componente y a nivel de aplicación hasta cierto grado.

Es posible que utilizar estados desactualizados pueda generar bugs al momento de utilizar objetos en vez de funciones en `setState()`. Realizamos traspasos de estado para compartirlos u ocultarlos de acuerdo a lo que sea necesario en distintos componentes. Algunas veces, un componente necesita realizar un traspaso de estado porque sus componentes hijos dependen de dicho estado. Quizá el componente está muy alejado del árbol de componentes, así que el estado debe ser accesible a lo largo de todo el árbol de componentes. Componentes están altamente involucrados en la gestión de estado, debido a que la principal responsabilidad de estos consiste en representar a la interfaz de usuario (UI).

Es por esto que hay soluciones que se encargan de la gestión de estado. Librerías como Redux[142] ó MobX[143] son ambas soluciones efectivas para implementar en aplicaciones React. Incluyen extensiones, react-redux[144] y mobx-react[145], para integrarlas en la capa de vistas de React. Redux y MobX están fuera del alcance de este libro, pero te animo a que estudies las diferentes maneras que existe para lidiar con la creciente necesidad por una óptima gestión de estado a medida que tus aplicaciones React se vuelven más y más complejas.

Ejercicios:

- Lee acerca de gestión externa de estado y cómo aprenderla[146]
- Revisa mi segundo libro acerca de gestión de estado en React llamado Taming the State in React (Domando el Estado en React)[147]

[142] https://redux.js.org/introduction
[143] https://mobx.js.org/
[144] https://github.com/reactjs/react-redux
[145] https://github.com/mobxjs/mobx-react
[146] https://www.robinwieruch.de/redux-mobx-confusion/
[147] https://roadtoreact.com/

¡Aprendiste sobre gestión de estado avanzada en React! Vamos a recapitular un poco:

- **React**
  - El traspaso de estado en React puede realizarse de dos maneras en componentes que así lo permitan
  - `setState()` puede utilizar una función para prevenir fallas o bugs generados por estados desactualizados
  - Existen distintas soluciones externas que pueden ayudarte a domar el estado en React

Puedes encontrar el código fuente en el repositorio oficial[148].

[148] https://github.com/the-road-to-learn-react/hackernews-client/tree/5.6

Pasos Finales hacia Producción

Los últimos capítulos te mostrarán cómo llevar tu aplicación a un estado de producción. Usarás el servicio de hospedaje gratuito Heroku. Mientras llevas tu aplicación a producción, aprenderás un poco más acerca de *create-react-app*.

Eject

El conocimiento que se presenta a continuación, **no es necesario** para llevar tu aplicación a producción. Sin embargo, vale la pena mencionarlo. *create-react-app* incluye una característica para prevenir la dependencia a un proveedor. La dependencia a un proveedor generalmente ocurre cuando has invertido en una tecnología y no hay forma de escapar de esta. En un caso de dependencia a proveedor es difícil cambiar la tecnología. Afortunadamente, en *create-react-app* tienes una ruta de escape con "eject".

En tu *package.json* encontrarás los scripts para *start*, *test* y *build* tu aplicación. El último script es *eject*. Lo puedes utilizar, *pero no hay forma de volver. **Es una operación en una sola dirección**. **Una vez que haces eject, ¡no hay vuelta atrás!**

Si acabas de comenzar a aprender React, no tiene sentido dejar el entorno conveniente de *create-react-app*.

Si te sientes lo suficientemente seguro como para ejecutar `npm run eject`, el comando copiará toda la configuración y dependencias a tu *package.json* y en un nuevo directorio *config/*. Aplicarás a todo el proyecto una configuración personalizada con herramientas que incluyen Babel, Webpack, y ESLint. Después de todo, tendrás control sobre todas estas herramientas.

La documentación oficial dice que *create-react-app* es adecuado para proyectos de tamaño pequeño a mediano. No debes sentir la obligación de user el comando *eject* a menos que te sientas preparado.

Ejercicios:

- lee más acerca de eject[149]

[149] https://github.com/facebookincubator/create-react-app#converting-to-a-custom-setup

Despliega tu Aplicación

Al final, ninguna aplicación se debe quedar en localhost. Querrás ponerla en línea. Heroku es una plataforma donde puedes hospedar tu aplicación. Ellos ofrecen una integración perfecta con React. Para ser más específico: Es posible desplegar una aplicación create-react-app en minutos. Es un despliegue que requiere cero configuración, por lo que sigue la filosofía de create-react-app.

Debes cumplir con dos requisitos antes de que sea posible desplegar tu aplicación en Heroku:

- Instalar el Heroku CLI[150]
- Crear una cuenta gratuita en Heroku[151]

Si has instalado Homebrew, puedes instalar el Heroku CLI desde la línea de comando:

Command Line

```
brew update
brew install heroku-toolbelt
```

Ahora puedes usar git y el Heroku CLI para hacer deploy de tu aplicación.

Command Line

```
git init
heroku create -b https://github.com/mars/create-react-app-buildpack.git
git add .
git commit -m "react-create-app on Heroku"
git push heroku master
heroku open
```

Eso es todo. Espero que tu aplicación esté ahora en funcionamiento. Si consigues problemas puedes consultar los siguientes recursos:

- Desplegando React con cero configuración[152]
- Heroku Buildpack para create-react-app[153]

[150] https://devcenter.heroku.com/articles/heroku-command-line
[151] https://www.heroku.com/
[152] https://blog.heroku.com/deploying-react-with-zero-configuration
[153] https://github.com/mars/create-react-app-buildpack

Esbozo

Llegamos al final de The Road to learn React. Espero que te haya ayudado ganar tracción con React. Si te gusto el libro compártelo con tus amigos como una manera de aprender React. Debería de ser utilizado como un regalo. Además, una reseña del libro en Amazon[154] o Goodreads[155] puede ayudar con proyectos futuros.

¿A donde ir ahora? Puedes extender la aplicación por tu cuenta o sumergirte en tu propio proyecto React. Antes de que te sumerjas en otro libro, curso o tutorial, deberías de crear un proyecto React con tus propias manos. Hazlo por una semana, llévalo a producción como se demostró en el ultimo capitulo, y contáctame en Twitter[156]. Tengo curiosidad de que construirás después de haber leído el libro. Tambien puedes encontrarme en GitHub[157] para compartir tu repositorio.

Si buscas expandir aún más tu aplicación, puedo recomendar varios caminos de aprendizaje:

- **Conectar la aplicación a una Base de Datos y/o Autenticación**: En una aplicación de React, quizá necesites implementar persistencia de datos. Los datos deben ser almacenados en una base de datos para que sobrevivan hasta luego del cierre de sesión del navegador y puedan ser compartidos entre diferentes usuarios utilizando distintas aplicaciones. La mejor manera de lograr esto es con Firebase. En este tutorial[158], encontrarás una guía paso-a-paso sobre como utilizar Firebase. Puedes utilizar Firebase para almacenar entidades de usuarios en tiempo real.
- **Gestión de Estado**: Has utilizado `this.setState()` para manejar el estado interno de los componentes. Es un comienzo perfecto. Sin embargo, en una aplicación en crecimiento experimentaras los limites del estado interno de los componentes. Por lo tanto tienes a tu alcance librerías de manejo de estado de terceros como Redux o MobX[159]. Mi siguiente libro[160] te servirá de guía en este tema.
- **Mecanización con WebPack y Babel**: En el libro utilizaste *create-react-app* para crear tu propia aplicación. En algún punto cuando hayas aprendido React, te convendría aprender la mecanización que lo rodea. Te permite preparar tus propios proyectos sin *create-react-app*. Puedo recomendar seguir una preparación mínima con Webpack y Babel[161]. Después podrías utilizar ESLint[162] para seguir un estilo unificado de código en tu aplicación.
- **Organización de Código**: En tu camino leyendo el libro te encontraste con un capítulo acerca de la Organización de Código. Puedes aplicar esos cambios ahora, si todavía no lo has hecho. Organizara tus componentes en archivos y carpetas estructuradas (módulos).

[154] https://www.amazon.com/dp/B077HJFCQX
[155] https://www.goodreads.com/book/show/33541539-the-road-to-learn-react
[156] https://twitter.com/rwieruch
[157] https://github.com/rwieruch
[158] https://www.robinwieruch.de/complete-firebase-authentication-react-tutorial/
[159] https://www.robinwieruch.de/redux-mobx-confusion/
[160] https://gumroad.com/products/uwiyI
[161] https://www.robinwieruch.de/minimal-react-webpack-babel-setup/
[162] https://www.robinwieruch.de/react-eslint-webpack-babel/

Además, te ayudara a comprender y aprender los principios de división de código, reusabilidad, mantenibilidad y diseño de módulos API.

- **Pruebas**: El libro solo araño la superficie de las pruebas. Si no eres familiar con el tema en general, puedes explorar en profundidad los conceptos de pruebas unitarias y de integración, especialmente en el contexto de aplicaciones React. En el nivel de implementación, recomiendo quedarse con Enzyme y Jest con el fin de refinar tu enfoque a las pruebas en React.
- **Sintaxis de Componentes React**: Las mejores prácticas para implementar componentes React están en constante evolución. En otros recursos encontrarás muchas maneras de escribir tus propios componente, especialmente componentes de clase React. El repositorio de GitHub llamado react-alternative-class-component-syntax[163] es un excelente recurso para conocer opciones alternativas para escribir componentes de clase React.
- **Componentes UI**: Muchos principiantes cometen el error de introducir librerías de componentes UI muy pronto en sus proyectos. Es mucho más práctico aprender a implementar y usar desplegables, checkboxes o diálogos en React por medio de elementos HTML estándar. La mayoría de estos componentes gestionan su propio estado a nivel local. Un checkbox necesita saber si esta chequeado o no, así que es necesario implementarlos como componentes controlados. Luego de haber conocido las implementaciones básicas, agregar una librería de componentes UI con checkboxes y diálogos como componente React debería resultar sencillo.
- **Enrutamiento**: Puedes aplicar enrutamiento en tu aplicación con react-router[164]. Hasta ahora solo tienes una página en tu aplicación. React Router te ayuda a tener múltiples páginas a través de distintos URLs.
- **Tipado**: En un capitulo utilizo React PropTypes para definir las interfaces de los componentes. Es una buena práctica realizada para prevenir errores. Pero los PropTypes solo son comprobados durante la ejecución. Puedes ir un paso más allá para introducir tipado estático el cual comprueba durante la compilación. TypeScript[165] es un enfoque popular. Pero en el ecosistema de React se usa frecuentemente Flow[166]. Yo recomiendo darle una oportunidad a Flow.
- **React Native**: React Native[167] trae tu aplicación a dispositivos móviles. Puedes aplicar tus conocimientos de React para enviar aplicaciones iOS y Android. La curva de aprendizaje, una vez hayas aprendido React, no debería de ser muy inclinada en React Native. Ambos comparten los mismos principios. Solo encontraras una disposición diferente de componentes en dispositivos móviles del que estás acostumbrado en aplicaciones web.
- **Otros Proyectos**: Hay muchos tutoriales que se enfocan solo en React para construir aplicaciones interesantes y que sirven de buena práctica para que apliques lo que aprendiste en este libro antes de que te muevas a proyectos intermedios. Aquí hay algunos de mi autoría:
  - A paginated and infinite scrolling list[168]
  - Showcasing tweets on a Twitter wall[169]
  - Connecting your React application to Stripe for charging money[170].

[163] https://github.com/the-road-to-learn-react/react-alternative-class-component-syntax
[164] https://github.com/ReactTraining/react-router
[165] https://www.typescriptlang.org/
[166] https://flowtype.org/
[167] https://facebook.github.io/react-native/
[168] https://www.robinwieruch.de/react-paginated-list/
[169] https://www.robinwieruch.de/react-svg-patterns/
[170] https://www.robinwieruch.de/react-express-stripe-payment/

Te invito a visitar mi sitio web[171] para encontrar más temas interesantes sobre desarrollo web, además de temas generales en ingeniería del software. Puedes suscribirte[172] para recibir nuevos artículos, libros y cursos. También puedes visitar la plataforma de Road to React[173] donde ofrezco lecciones más avanzadas sobre el ecosistema React.

Finalmente, espero encontrar más Patrons[174] dispuestos a patrocinar mi contenido, hay muchos estudiantes que no pueden pagar por contenido educacional. Por eso, publico gran parte de mi contenido gratuitamente. Patrocinarme en Patreon permite que otros tengan acceso a este contenido de forma gratuita.

Una vez más, si te gustó el libro, quiero que te tomes un momento de pensar en una persona interesada en React en aprender React. Acércate a esa persona y comparte el libro. Esto significaría mucho para mí. El propósito de este libro es el ser dado a otros. Mejorará con el tiempo, cuando más personas lo utilicen y compartan sus comentarios.

Muchas gracias por leer The Road to learn React.

Saludos,

Robin Wieruch

[171] https://www.robinwieruch.de/
[172] https://www.getrevue.co/profile/rwieruch
[173] https://roadtoreact.com
[174] https://www.patreon.com/rwieruch